图灵程序设计丛书

MongoDB: The Definitive Guide

第2版

MongoDB 权威指南

[美] *Kristina Chodorow* 著
邓强 王明辉 译

O'REILLY®


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍

邓强

软件工程师，常年关注互联网行业，对互联网产品和技术兴趣浓厚，Apple粉、Google粉、Amazon粉。职业生涯的前几年一直从事金融行业软件系统开发，后来不顾一切投身互联网。目前在一个新成立的互联网创业小团队任职。Email：dengqiang@outlook.com。

王明辉

同济大学计算机系学生，经常使用MongoDB。知乎网常用id为纳米黑客。

 图灵程序设计丛书

MongoDB权威指南（第2版）

MongoDB: The Definitive Guide
Second Edition

［美］Kristina Chodorow 著
邓强　王明辉 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

MongoDB权威指南 : 第2版 / (美) 霍多罗夫
(Chodorow,K.) 著 ; 邓强, 王明辉译. -- 北京 : 人民
邮电出版社, 2014.1(2016.9 重印)
（图灵程序设计丛书）
书名原文: MongoDB:the definitive guide, second
edition
ISBN 978-7-115-34108-2

Ⅰ. ①M… Ⅱ. ①霍… ②邓… ③王… Ⅲ. ①关系数
据库系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2013)第302154号

内 容 提 要

与传统的关系型数据库不同，MongoDB 是一种面向文档的数据库。本书这一版共分为六部分，涵盖开发、管理以及部署的各个方面。第一部分展示 MongoDB 基础知识、核心概念。第二部分介绍使用 MongoDB 进行开发，包括索引的概念以及各种特殊索引和集合的用法等。第三部分讲述复制，包括副本集的相关概念、创建方法，与应用程序的交互等。第四部讨论分片，包括分片的配置，片键的选择，集群的管理。第五部分阐述创建索引、移动和压缩数据等管理任务，以及 MongoDB 的持久数据存储。最后一部分集中说明服务器管理。

本书适合数据库开发和管理人员阅读。

◆ 著　　　　[美] Kristina Chodorow
译　　　　邓　强　王明辉
责任编辑　李松峰
执行编辑　李　静
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　　印刷

◆ 开本：800×1000　1/16
印张：25.75
字数：490千字　　　　　　2014年 1 月第 1 版
印数：19 001 – 20 500册　　2016年 9 月北京第 10 次印刷

著作权合同登记号　图字：01-2013-6738号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

第二部分　设计应用

第三部分　复制

第四部分　分片

第五部分　应用管理

第六部分　服务器管理

序

10 年前，没人能预见互联网的发展会给关系型数据库带来如此多的挑战。在此期间，我亲身经历了在快速发展的大型互联网公司应用 MySQL 的过程。开始时只有很少的数据，一台服务器就可以了。然后就得建立备份，以便应对大量的读取和不时的宕机。用不了多长时间，就得加一个缓存层，调整所有的查询，投入更多的硬件。

最后，你会发现自己需要将数据切分到多个集群上，并重新构建大量的应用逻辑以适应这种切分。之后不久，你又会发现被自己数月前设计的数据库结构限制住了。

怎么会呢？这是因为现在集群中的数据太多，需要更改模式，会花费很长时间，也需要 DBA 投入相当多的宝贵时间。在代码中处理要简单一些，但也需要小型开发团队数月的努力。最后，你会不断地拷问自己有没有更好的方法，或者为什么没有在核心数据库服务器中内置更多此类功能。

为了应对现在 Web 应用的数据膨胀，开源社区像以往一样提供了太多的“好方法”。从内存中的键值型存储到可以使用 SQL 的 MySQL/InnoDB 变种等复杂方法，无所不有。但选择多了，做出正确的选择反而更难了。我自己就研究过其中很多种。

MongoDB 的实用性着实令人着迷。MongoDB 并不去迎合所有人的全部需求。它在功能和复杂性之间取得了很好的平衡，并且大大简化了原先十分复杂的任务。也就是说，它具备支撑今天主流 Web 应用的关键功能：索引、复制、分片、丰富的查询语法，特别灵活的数据模型。与此同时还不牺牲速度。

秉持 MongoDB 自身的风格，本书简洁明快、通俗易懂。MongoDB 新用户通过阅读第 1 章，马上就能入门，而有经验的用户则可以体验到本书的广度和权威性。对于流行的客户端 API 和高级的管理主题，如复制、备份和分片，本书都是权威参考。

根据我最近每天使用 MongoDB 的经验，我相信本书会始终不离我左右，从最初安装到进行分片或备份式集群的产品化部署，它都是我最好的助手。任何想仔细研究使用 MongoDB 的人都需要这本重要的参考书。

——Craigslist 软件工程师，Jeremy Zawodny

2010 年 8 月

前言

本书的组织结构

本书分为六个部分，涵盖了开发、管理以及部署的方方面面。

熟悉MongoDB

第 1 章将简要讲述 MongoDB 的背景：项目创立原因，希望达到的目标，选用它的理由。第 2 章接着介绍一些 MongoDB 的核心概念和术语，还有如何上手操作数据库和 shell 的相关内容。接下来两章介绍 MongoDB 开发者需要掌握的基础知识。第 3 章展示如何执行基本的写入操作，包括在不同安全和速度等级下的实现细节。第 4 章主要介绍如何查找文档和创建复杂的查询。这一章还包括如何迭代结果集和其他一些用于处理结果集的方法，比如限制结果集的数量，略过一些结果，以及对结果集排序。

使用MongoDB进行开发

第 5 章将介绍什么是索引以及如何为 MongoDB 的集合建立索引。第 6 章说明如何使用各种特殊类型的索引和集合。第 7 章展示一些利用 MongoDB 聚集数据的方法，包括计数、查找唯一值、文档分组、聚合框架和 MapReduce。这一部分的最后一章会介绍如何设计应用程序：第 8 章讲述如何更好地在应用程序中使用 MongoDB。

复制

第 9 章开始介绍复制，着重讲述如何快速在本地建立一个副本集，还会介绍一些可用选项。第 10 章涵盖了与副本集相关的一些概念。第 11 章展示了副本集与应用程序的交互。第 12 章从管理的角度介绍副本集的运行。

分片

第 13 章开始介绍分片，并通过一个例子展示如何快速地在本地进行分片。第 14 章介绍集群的组成以及设置。第 15 章介绍如何为不同的应用程序选择合适的片键。最后，第 16 章介绍分片集群的管理。

应用程序管理

接下来两章从应用程序的角度介绍 MongoDB 管理的很多方面。第 17 章讲述如何查看 MongoDB 正在进行的操作。第 18 章介绍一些管理任务，比如创建索引、移动和压缩数据。第 19 章介绍 MongoDB 的持久数据存储。

服务器管理

最后一部分集中介绍服务器管理。第 20 章将给出启动和终止 MongoDB 时的一些通用选项。第 21 章讨论在监控数据库运行时如何查看监控信息。第 22 章介绍在不同类型的部署中如何备份和恢复数据库。最后，第 23 章将介绍部署 MongoDB 时需要牢记于心的一些系统设置。

附录

附录 A 介绍了 MongoDB 的版本控制方案，以及在 Windows、OS X 和 Linux 上的安装细节。附录 B 详细说明了 MongoDB 的内部工作原理：存储引擎、数据格式和传输协议。

本书排版规范

本书使用的排版规范如下所示。

- 楷体

  用于表示新的术语。

- 等宽字体

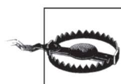
  表示程序片段，也在段落中表示程序中使用的变量、函数名、命令行实用工具、环境变量、语句和关键字等元素。

- 等宽斜体

  用户需要根据自己提供的值或由上下文确定的值进行更改的部分。



这个图标代表小窍门、建议或说明。



这个图标代表警告信息。

使用代码示例

让本书助你一臂之力。也许你要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。例如，无需请求许可，就可以用本书中的几段代码写成一个程序。但是销售或者发布 O'Reilly 图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

我们非常感谢你能标明出处，但并不强求。出处一般包括书名、作者、出版商和 ISBN，例如《MongoDB 权威指南（第 2 版）》，Kristina Chodorow 著（O'Reilly，2013）。版权所有，978-1-449-34468-9。

如果有关于使用代码的未尽事宜，可以随时与我们联系：permissions@oreilly.com。

Safari在线图书



Safari 在线图书（www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版 世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 SafariBooks Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/mongodb_2e

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

感谢本书的技术审稿人 Adam Comerford、Eric Milke 和 Greg Studer。他们在完善本书的过程中做出了不懈努力。感谢优秀的编辑 Ann Spencer 在本书出版的每一阶段对我的帮助。感谢我在 10gen 的各位同事，感谢他们与我分享 MongoDB 的知识和建议，同时也感谢 Eliot Horowitz 和 Dwight Merriman 启动 MongoDB 这个项目。最后要感谢 Andrew 在本书编写过程中提供的支持和建议。

第一部分

MongoDB介绍

第 1 章

MongoDB简介

MongoDB 是一款强大、灵活，且易于扩展的通用型数据库。它能扩展出非常多的功能，如二级索引（secondary index）、范围查询（range query）、排序、聚合（aggregation），以及地理空间索引（geospatial index）。本章涵盖了 MongoDB 的主要设计特点。

1.1 易于使用

MongoDB 是一个面向文档（document-oriented）的数据库，而不是关系型数据库。不采用关系模型主要是为了获得更好的扩展性。当然，还有其他一些好处。

与关系型数据库相比，面向文档的数据库不再有“行”（row）的概念，取而代之的是更为灵活的“文档”（document）模型。通过在文档中嵌入文档和数组，面向文档的方法能够仅使用一条记录来表现复杂的层次关系，这与使用现代面向对象语言的开发者对数据的看法一致。

另外，不再有预定义模式（predefined schema）：文档的键（key）和值（value）不再是固定的类型和大小。由于没有固定的模式，根据需要添加或删除字段变得更容易了。通常，由于开发者能够进行快速迭代，所以开发进程得以加快。而且，实验更容易进行。开发者能尝试大量的数据模型，从中选择一个最好的。

1.2 易于扩展

应用程序数据集的大小正在以不可思议的速度增长。随着可用带宽的增长和存储器价格的下降，即使是一个小规模的应用程序，需要存储的数据量也可能大得惊人，甚至超出了很多数据库的处理能力。过去非常罕见的 T 级别数据，现在已是司空见惯了。

由于需要存储的数据量不断增长，开发者面临一个困难：应该如何扩展数据库？实质上，这是纵向扩展（scale up）和横向扩展（scale out）之间的选择。纵向扩展就是使用计算能力更强的机器，而横向扩展就是通过分区将数据分散到更多机器上。通常，纵向扩展是最省力的做法，其缺点是大型机一般都非常昂贵。而且，当数据量达到机器的物理极限时，无论花多少钱也买不到更强的机器了。另一个选择是横向扩展：要增加存储空间或提高性能，只需购买一台普通的服务器并把它添加到集群中就可以了。横向扩展既便宜又易于扩展；不过，管理 1000 台机器比管理一台机器显然要困难得多。

MongoDB 的设计采用横向扩展。面向文档的数据模型使它能很容易地在多台服务器之间进行数据分割。MongoDB 能自动处理跨集群的数据和负载，自动重新分配文档，以及将用户请求路由到正确的机器上。这样，开发者能够集中精力编写应用程序，而不需要考虑如何扩展的问题。如果一个集群需要更大的容量，只需要向集群添加新服务器，MongoDB 就会自动将现有数据向新服务器传送。

1.3 丰富的功能

MongoDB 作为一款通用型数据库，除了能够创建、读取、更新和删除数据之外，还提供一系列不断扩展的独特功能。

- 索引（indexing）
  MongoDB 支持通用二级索引，允许多种快速查询，且提供唯一索引、复合索引、地理空间索引，以及全文索引。

- 聚合（aggregation）
  MongoDB 支持“聚合管道”（aggregation pipeline）。用户能通过简单的片段创建复杂的聚合，并通过数据库自动优化。

- 特殊的集合类型
  MongoDB 支持存在时间有限的集合，适用于那些将在某个时刻过期的数据，如会话（session）。类似地，MongoDB 也支持固定大小的集合，用于保存近期数据，如日志。

- 文件存储（file storage）
  MongoDB 支持一种非常易用的协议，用于存储大文件和文件元数据。

MongoDB 并不具备一些在关系型数据库中很普遍的功能，如连接（join）和复杂的多行事务（multirow transaction）。省略这些功能是出于架构上的考虑（为了得到更好的扩展性），因为在分布式系统中这两个功能难以高效地实现。

1.4 卓越的性能

MongoDB 的一个主要目标是提供卓越的性能，这很大程度上决定了 MongoDB 的设计。MongoDB 能对文档进行动态填充（dynamic padding），也能预分配数据文件以利用额外的空间来换取稳定的性能。MongoDB 把尽可能多的内存用作缓存（cache），试图为每次查询自动选择正确的索引。总之，MongoDB 在各方面的设计都旨在保持它的高性能。

虽然，MongoDB 非常强大并试图保留关系型数据库的很多特性，但它并不追求具备关系型数据库的所有功能。只要有可能，数据库服务器就会将处理和逻辑交给客户端（通过驱动程序或用户的应用程序代码来实现）。这种精简方式的设计是 MongoDB 能够实现如此高性能的原因之一。

1.5 小结

本书将详细说明 MongoDB 开发过程中的一些特定设计背后的原因和动机，借此分享 MongoDB 背后的哲学。当然，掌握 MongoDB 最好的方式是创建一个易扩展、灵活、快速的功能完备的数据存储，这也是 MongoDB 的意义所在。

第2章
MongoDB基础知识

MongoDB 非常强大但很容易上手。本章会介绍一些 MongoDB 的基本概念。

- 文档是 MongoDB 中数据的基本单元，非常类似于关系型数据库管理系统中的行，但更具表现力。
- 类似地，集合（collection）可以看作是一个拥有动态模式（dynamic schema）的表。
- MongoDB 的一个实例可以拥有多个相互独立的数据库（database），每一个数据库都拥有自己的集合。
- 每一个文档都有一个特殊的键 "_id"，这个键在文档所属的集合中是唯一的。
- MongoDB 自带了一个简单但功能强大的 JavaScript shell，可用于管理 MongoDB 的实例或数据操作。

2.1 文档

文档是 MongoDB 的核心概念。文档就是键值对的一个有序集。每种编程语言表示文档的方法不太一样，但大多数编程语言都有一些相通的数据结构，比如映射（map）、散列（hash）或字典（dictionary）。例如，在 JavaScript 里面，文档被表示为对象：

```
{"greeting" : "Hello, world!"}
```

这个文档只有一个键 "greeting"，其对应的值为 "Hello，world!"。大多数文档会比这个简单的例子复杂得多，通常会包含多个键 / 值对：

```
{"greeting" : "Hello, world!", "foo" : 3}
```

从上面的例子可以看出，文档中的值可以是多种不同的数据类型（甚至可以是一个完整的内嵌文档，详见 2.6.4 节）。在这个例子中，"greeting" 的值是一个字符串，而 "foo" 的值是一个整数。

文档的键是字符串。除了少数例外情况，键可以使用任意 UTF-8 字符。

- 键不能含有 \0（空字符）。这个字符用于表示键的结尾。
- . 和 $ 具有特殊意义，只能在特定环境下使用（后面的章节会详细说明）。通常，这两个字符是被保留的；如果使用不当的话，驱动程序会有提示。

MongoDB 不但区分类型，而且区分大小写。例如，下面的两个文档是不同的：

```
{"foo" : 3}
{"foo" : "3"}
```

下面两个文档也是不同的：

```
{"foo" : 3}
{"Foo" : 3}
```

还有一个非常重要的事项需要注意，MongoDB 的文档不能有重复的键。例如，下面的文档是非法的：

```
{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}
```

文档中的键 / 值对是有序的：{"x" : 1, "y"：2} 与 {"y": 2, "x": 1} 是不同的。通常，字段顺序并不重要，无须让数据库模式依赖特定的字段顺序（MongoDB 会对字段重新排序）。在某些特殊情况下，字段顺序变得非常重要，本书将就此给出提示。

一些编程语言对文档的默认表示根本就不包含顺序问题（如：Python 中的字典、Perl 和 Ruby 1.8 中的散列）。通常，这些语言的驱动具有某些特殊的机制，可以在必要时指定文档的顺序。

2.2　集合

集合就是一组文档。如果将 MongoDB 中的一个文档比喻为关系型数据库中的一行，那么一个集合就相当于一张表。

2.2.1　动态模式

集合是动态模式的。这意味着一个集合里面的文档可以是各式各样的。例如，下面两个文档可以存储在同一个集合里面：

```
{"greeting" : "Hello, world!"}
{"foo" : 5}
```

需要注意的是，上面的文档不光值的类型不同（一个是字符串，一个是整数），它们的键也完全不同。因为集合里面可以放置任何文档，随之而来的一个问题是：还有必要使用多个集合吗？这的确值得思考：既然没有必要区分不同类型文档的模式，为什么还要使用多个集合呢？这里有几个重要的原因。

- 如果把各种各样的文档不加区分地放在同一个集合里，无论对开发者还是对管理员来说都将是噩梦。开发者要么确保每次查询只返回特定类型的文档，要么让执行查询的应用程序来处理所有不同类型的文档。如果查询博客文章时还要剔除含有作者数据的文档，这会带来很大困扰。
- 在一个集合里查询特定类型的文档在速度上也很不划算，分开查询多个集合要快得多。例如，假设集合里面一个名为 `"type"` 的字段用于指明文档是 skim、whole 还是 chunky monkey。那么，如果从一个集合中查询这三种类型的文档，速度会很慢。但如果将这三种不同类型的文档拆分为三个不同的集合，每次只需要查询相应的集合，速度快得多。
- 把同种类型的文档放在一个集合里，数据会更加集中。从一个只包含博客文章的集合里查询几篇文章，或者从同时包含文章数据和作者数据的集合里查出几篇文章，相比之下，前者需要的磁盘寻道操作更少。
- 创建索引时，需要使用文档的附加结构（特别是创建唯一索引时）。索引是按照集合来定义的。在一个集合中只放入一种类型的文档，可以更有效地对集合进行索引。

上面这些重要原因促使我们创建一个模式，把相关类型的文档组织在一起，尽管 MongoDB 对此并没有强制要求。

2.2.2　命名

集合使用名称进行标识。集合名可以是满足下列条件的任意 UTF-8 字符串。

- 集合名不能是空字符串（""）。
- 集合名不能包含 \0 字符（空字符），这个字符表示集合名的结束。
- 集合名不能以“system.”开头，这是为系统集合保留的前缀。例如，system.users 这个集合保存着数据库的用户信息，而 system.namespaces 集合保存着所有数据库集合的信息。
- 用户创建的集合不能在集合名中包含保留字符 `'$'`。因为某些系统生成的集合中包含 `$`，很多驱动程序确实支持在集合名里包含该字符。除非你要访问这种系统创建的集合，否则不应该在集合名中包含 `$`。

子集合

组织集合的一种惯例是使用“.”分隔不同命名空间的子集合。例如，一个具有博客功能的应用可能包含两个集合，分别是 blog.posts 和 blog.authors。这是为了使组织结构更清晰，这里的 blog 集合（这个集合甚至不需要存在）跟它的子集合没有任何关系。

虽然子集合没有任何特别的属性，但它们却非常有用，因而很多 MongoDB 工具都使用了子集合。

- GridFS（一种用于存储大文件的协议）使用子集合来存储文件的元数据，这样就可以与文件内容块很好地隔离开来。（第 6 章会详细介绍 GridFS。）
- 大多数驱动程序都提供了一些语法糖，用于访问指定集合的子集合。例如，在数据库 shell 中，`db.blog` 代表 blog 集合，而 `db.blog.posts` 代表 blog.posts 集合。

在 MongoDB 中，使用子集合来组织数据非常高效，值得推荐。

2.3　数据库

在 MongoDB 中，多个文档组成集合，而多个集合可以组成*数据库*。一个 MongoDB 实例可以承载多个数据库，每个数据库拥有 0 个或者多个集合。每个数据库都有独立的权限，即便是在磁盘上，不同的数据库也放置在不同的文件中。按照经验，我们将有关一个应用程序的所有数据都存储在同一个数据库中。要想在同一个 MongoDB 服务器上存放多个应用程序或者用户的数据，就需要使用不同的数据库。

数据库通过名称来标识，这点与集合类似。数据库名可以是满足以下条件的任意 UTF-8 字符串。

- 不能是空字符串（""）。
- 不得含有 /、\、.、"、*、<、>、:、|、?、$、（一个空格）、\0（空字符）。基本上，只能使用 ASCII 中的字母和数字。
- 数据库名区分大小写，即便是在不区分大小写的文件系统中也是如此。简单起见，数据库名应全部小写。
- 数据库名最多为 64 字节。

要记住一点，数据库最终会变成文件系统里的文件，而数据库名就是相应的文件名，这是数据库名有如此多限制的原因。

另外，有一些数据库名是保留的，可以直接访问这些有特殊语义的数据库。这些数

据库如下所示。

- admin
  从身份验证的角度来讲，这是“root”数据库。如果将一个用户添加到 admin 数据库，这个用户将自动获得所有数据库的权限。再者，一些特定的服务器端命令也只能从 admin 数据库运行，如列出所有数据库或关闭服务器。

- local
  这个数据库永远都不可以复制，且一台服务器上的所有本地集合都可以存储在这个数据库中。（第 9 章会详细介绍复制及本地数据库。）

- config
  MongoDB 用于分片设置时（参见第 13 章），分片信息会存储在 config 数据库中。

把数据库名添加到集合名前，得到集合的完全限定名，即命名空间（namespace）。例如，如果要使用 cms 数据库中的 blog.posts 集合，这个集合的命名空间就是 `cms.blog.posts`。命名空间的长度不得超过 121 字节，且在实际使用中应小于 100 字节。（参考附录 B，了解 MongoDB 中集合的命名空间及内部表示的更多信息。）

2.4 启动MongoDB

通常，MongoDB 作为网络服务器来运行，客户端可连接到该服务器并执行操作。下载 MongoDB（http://www.mongodb.org/downloads）并解压，运行 `mongod` 命令，启动数据库服务器：

```
$ mongod
mongod --help for help and startup options
Thu Oct 11 12:36:48 [initandlisten] MongoDB starting : pid=2425 port=27017
    dbpath=/data/db/ 64-bit host=spock
Thu Oct 11 12:36:48 [initandlisten] db version v2.4.0, pdÑle version 4.5
Thu Oct 11 12:36:48 [initandlisten] git version:
    3aaea5262d761e0bb6bfef5351cfbfca7af06ec2
Thu Oct 11 12:36:48 [initandlisten] build info: Darwin spock 11.2.0 Darwin Kernel
    Version 11.2.0: Tue Aug 9 20:54:00 PDT 2011;
    root:xnu-1699.24.8~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_48
Thu Oct 11 12:36:48 [initandlisten] options: {}
Thu Oct 11 12:36:48 [initandlisten] journal dir=/data/db/journal
Thu Oct 11 12:36:48 [initandlisten] recover : no journal Ñles present, no
    recovery needed
Thu Oct 11 12:36:48 [websvr] admin web console waiting for connections on
    port 28017
Thu Oct 11 12:36:48 [initandlisten] waiting for connections on port 27017
```

在 Windows 系统中，执行这个命令：

```
$ mongod.exe
```



关于安装 MongoDB 的详细信息，参见附录 A。

mongod 在没有参数的情况下会使用默认数据目录 /data/db（Windows 系统中为 C:\data\db）。如果数据目录不存在或者不可写，服务器会启动失败。因此，在启动 MongoDB 前，先创建数据目录（如 mkdir -p /data/db/），以确保对该目录有写权限，这点非常重要。

启动时，服务器会打印版本和系统信息，然后等待连接。默认情况下，MongoDB 监听 27017 端口。如果端口被占用，启动将失败。通常，这是由于已经有一个 MongoDB 实例在运行了。

mongod 还会启动一个非常基本的 HTTP 服务器，监听数字比主端口号高 1000 的端口，也就是 28017 端口。这意味着，通过浏览器访问 http://localhost:28017，能获取数据库的管理信息。

中止 mongod 的运行，只须在运行着服务器的 shell 中按下 Ctrl-C。



要想了解启动和停止 MongoDB 的更多细节，参见第 20 章。

2.5 MongoDB shell简介

MongoDB 自带 JavaScript shell，可在 shell 中使用命令行与 MongoDB 实例交互。shell 非常有用，通过它可以执行管理操作，检查运行实例，亦或做其他尝试。对 MongoDB 来说，mongo shell 是至关重要的工具，其应用之广泛将体现在本书接下来的部分中。

2.5.1 运行shell

运行 mongo 启动 shell：

```
$ mongo
MongoDB shell version: 2.4.0
connecting to: test
>
```

启动时，shell 将自动连接 MongoDB 服务器，须确保 mongod 已启动。

shell 是一个功能完备的 JavaScript 解释器，可运行任意 JavaScript 程序。为说明这一点，我们运行几个简单的数学运算：

```
> x = 200
200
> x / 5;
40
```

另外，可充分利用 JavaScript 的标准库：

```
> Math.sin(Math.PI / 2);
1
> new Date("2010/1/1");
"Fri Jan 01 2010 00:00:00 GMT-0500 (EST)"
> "Hello, World!".replace("World", "MongoDB");
Hello, MongoDB!
```

再者，可定义和调用 JavaScript 函数：

```
> function factorial (n) {
... if (n <= 1) return 1;
... return n * factorial(n - 1);
... }
> factorial(5);
120
```

需要注意，可使用多行命令。shell 会检测输入的 JavaScript 语句是否完整，如没写完可在下一行接着写。在某行连续三次按下回车键可取消未输入完成的命令，并退回到 > －提示符。

2.5.2 MongoDB客户端

能运行任意 JavaScript 程序听上去很酷，不过 shell 的真正强大之处在于，它是一个独立的 MongoDB 客户端。启动时，shell 会连到 MongoDB 服务器的 test 数据库，并将数据库连接赋值给全局变量 db。这个变量是通过 shell 访问 MongoDB 的主要入口点。

如果想要查看 db 当前指向哪个数据库，可以使用 db 命令：

```
> db
test
```

为了方便习惯使用 SQL shell 的用户，shell 还包含一些非 JavaScript 语法的扩展。这些扩展并不提供额外的功能，而是一些非常棒的语法糖。例如，最重要的操作之一为选择数据库：

```
> use foobar
switched to db foobar
```

现在，如果查看 `db` 变量，会发现其正指向 foobar 数据库：

```
> db
foobar
```

因为这是一个 JavaScript shell，所以键入一个变量会将此变量的值转换为字符串（即数据库名）并打印出来。

通过 `db` 变量，可访问其中的集合。例如，通过 `db.baz` 可返回当前数据库的 baz 集合。因为通过 shell 可访问集合，这意味着，几乎所有数据库操作都可以通过 shell 完成。

2.5.3　shell中的基本操作

在 shell 中查看或操作数据会用到 4 个基本操作：创建、读取、更新和删除（即通常所说的 CRUD 操作）。

1. 创建

`insert` 函数可将一个文档添加到集合中。举一个存储博客文章的例子。首先，创建一个名为 `post` 的局部变量，这是一个 JavaScript 对象，用于表示我们的文档。它会有几个键：`"title"`、`"content"` 和 `"date"`（发布日期）。

```
> post = {"title" : "My Blog Post",
... "content" : "Here's my blog post.",
... "date" : new Date()}
{
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

这个对象是个有效的 MongoDB 文档，所以可以用 `insert` 方法将其保存到 blog 集合中：

```
> db.blog.insert(post)
```

这篇文章已被存到数据库中。要查看它可用调用集合的 `find` 方法：

```
> db.blog.find()
{
    "_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
```

```
    "date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

可以看到，我们曾输入的键 / 值对都已被完整地记录。此外，还有一个额外添加的键 "_id"。"_id" 突然出现的原因将在本章末尾解释。

2. 读取

find 和 findOne 方法可以用于查询集合里的文档。若只想查看一个文档，可用 findOne：

```
> db.blog.findOne()
{
    "_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

find 和 findOne 可以接受一个查询文档作为限定条件。这样就可以查询符合一定条件的文档。使用 find 时，shell 会自动显示最多 20 个匹配的文档，也可获取更多文档。第 4 章会详细介绍查询相关的内容。

3. 更新

使用 update 修改博客文章。update 接受（至少）两个参数：第一个是限定条件（用于匹配待更新的文档），第二个是新的文档。假设我们要为先前写的文章增加评论功能，就需要增加一个新的键，用于保存评论数组。

首先，修改变量 post，增加 "comments" 键：

```
> post.comments = []
[ ]
```

然后执行 update 操作，用新版本的文档替换标题为“My Blog Post”的文章：

```
> db.blog.update({title : "My Blog Post"}, post)
```

现在，文档已经有了 "comments" 键。再用 find 查看一下，可以看到新的键：

```
> db.blog.find()
{
    "_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : ISODate("2012-08-24T21:12:09.982Z"),
    "comments" : [ ]
}
```

4. 删除

使用 remove 方法可将文档从数据库中永久删除。如果没有使用任何参数，它会将集合内的所有文档全部删除。它可以接受一个作为限定条件的文档作为参数。例如，下面的命令会删除刚刚创建的文章：

```
> db.blog.remove({title : "My Blog Post"})
```

现在，集合又是空的了。

2.6 数据类型

本章开始部分介绍了文档的基本概念，现在你已经会启动、运行 MongoDB，也会在 shell 中进行一些操作了。这一节的内容会更加深入。MongoDB 支持将多种数据类型作为文档中的值，下面将一一介绍。

2.6.1 基本数据类型

在概念上，MongoDB 的文档与 JavaScript 中的对象相近，因而可认为它类似于 JSON。JSON（http://www.json.org）是一种简单的数据表示方式：其规范仅用一段文字就能描述清楚（其官网证明了这点），且仅包含 6 种数据类型。这样有很多好处：易于理解、易于解析、易于记忆。然而，从另一方面来说，因为只有 null、布尔、数字、字符串、数组和对象这几种数据类型，所以 JSON 的表达能力有一定的局限。

虽然 JSON 具备的这些类型已具有很强的表现力，但绝大多数应用（尤其是在与数据库打交道时）都还需要其他一些重要的类型。例如，JSON 没有日期类型，这使原本容易的日期处理变得烦人。另外，JSON 只有一种数字类型，无法区分浮点数和整数，更别说区分 32 位和 64 位数字了。再者，JSON 无法表示其他一些通用类型，如正则表达式或函数。

MongoDB 在保留 JSON 基本键 / 值对特性的基础上，添加了其他一些数据类型。在不同的编程语言下，这些类型的确切表示有些许差异。下面说明 MongoDB 支持的其他通用类型，以及如何在文档中使用它们。

- null

  null 用于表示空值或者不存在的字段：

  ```
  {"x" : null}
  ```

- 布尔型

  布尔类型有两个值 true 和 false：

```
{"x" : true}
```

- 数值

  shell 默认使用 64 位浮点型数值。因此，以下数值在 shell 中是很“正常”的：

  ```
  {"x" : 3.14}
  ```

  或：

  ```
  {"x" : 3}
  ```

  对于整型值，可使用 `NumberInt` 类（表示 4 字节带符号整数）或 `NumberLong` 类（表示 8 字符带符号整数），分别举例如下：

  ```
  {"x" : NumberInt("3")}
  {"x" : NumberLong("3")}
  ```

- 字符串

  UTF-8 字符串都可表示为字符串类型的数据：

  ```
  {"x" : "foobar"}
  ```

- 日期

  日期被存储为自新纪元以来经过的毫秒数，不存储时区：

  ```
  {"x" : new Date()}
  ```

- 正则表达式

  查询时，使用正则表达式作为限定条件，语法也与 JavaScript 的正则表达式语法相同：

  ```
  {"x" : /foobar/i}
  ```

- 数组

  数据列表或数据集可以表示为数组：

  ```
  {"x" : ["a", "b", "c"]}
  ```

- 内嵌文档

  文档可嵌套其他文档，被嵌套的文档作为父文档的值：

  ```
  {"x" : {"foo" : "bar"}}
  ```

- 对象id

  对象 id 是一个 12 字节的 ID，是文档的唯一标识。详见 2.6.5 节。

  ```
  {"x" : ObjectId()}
  ```

还有一些不那么常用，但可能有需要的类型，包括下面这些。

- 二进制数据

  二进制数据是一个任意字节的字符串。它不能直接在 shell 中使用。如果要将非 UTF-8 字符保存到数据库中，二进制数据是唯一的方式。

- 代码

  查询和文档中可以包括任意 JavaScript 代码：

  ```
  {"x" : function() { /* ... */ }}
  ```

另外，有几种大多数情况下仅在内部使用（或被其他类型取代）的类型。在本书中，出现这种情况时会特别说明。

关于 MongoDB 数据格式的更多信息，参考附录 B。

2.6.2 日期

在 JavaScript 中，`Date` 类可以用作 MongoDB 的日期类型。创建日期对象时，应使用 `new Date`（…），而非 `Date`（…）。如将构造函数（constructor）作为函数进行调用（即不包括 `new` 的方式），返回的是日期的字符串表示，而非日期（`Date`）对象。这个结果与 MongoDB 无关，是 JavaScript 的工作机制决定的。如果不注意这一点，没有始终使用日期（`Date`）构造函数，将得到一堆混乱的日期对象和日期的字符串。由于日期和字符串之间无法匹配，所以执行删除、更新及查询等几乎所有操作时会导致很多问题。

关于 JavaScript 日期类的完整解释，以及构造函数的参数格式，参见 ECMAScript 规范 15.9 节（http://www.ecmascript.org）。

shell 根据本地时区设置显示日期对象。然而，数据库中存储的日期仅为新纪元以来的毫秒数，并未存储对应的时区。（当然，可将时区信息存储为另一个键的值）。

2.6.3 数组

数组是一组值，它既能作为有序对象（如列表、栈或队列），也能作为无序对象（如数据集）来操作。

在下面的文档中，`"things"` 这个键的值是一个数组：

```
{"things" : ["pie", 3.14]}
```

此例表示，数组可包含不同数据类型的元素（在此，是一个字符串和一个浮点数）。

实际上，常规的键 / 值对支持的所有值都可以作为数组的值，数组中甚至可以套嵌数组。

文档中的数组有个奇妙的特性，就是 MongoDB 能“理解”其结构，并知道如何“深入”数组内部对其内容进行操作。这样就能使用数组内容对数组进行查询和构建索引了。例如，之前的例子中，MongoDB 可以查询出 `"things"` 数组中包含 3.14 这个元素的所有文档。要是经常使用这个查询，可以对 `"things"` 创建索引来提高性能。

MongoDB 可以使用原子更新对数组内容进行修改，比如深入数组内部将 pie 改为 pi。本书后面还会介绍更多这种操作的例子。

2.6.4 内嵌文档

文档可以作为键的值，这样的文档就是*内嵌文档*。使用内嵌文档，可以使数据组织方式更加自然，不用非得存成扁平结构的键 / 值对。

例如，用一个文档来表示一个人，同时还要保存他的地址，可以将地址信息保存在内嵌的 `"address"` 文档中：

```
{
    "name" : "John Doe",
    "address" : {
        "street" : "123 Park Street",
        "city" : "Anytown",
        "state" : "NY"
    }
}
```

上面例子中 `"address"` 键的值是一个内嵌文档，这个文档有自己的 `"street"`、`"city"` 和 `"state"` 键以及对应的值。

同数组一样，MongoDB 能够“理解”内嵌文档的结构，并能“深入”其中构建索引、执行查询或者更新。

稍后会深入讨论模式设计，但是从这个简单的例子也可以看得出内嵌文档可以改变处理数据的方式。在关系型数据库中，这个例子中的文档一般会被拆分成两个表中的两个行（“people”和“address”各一行）。在 MongoDB 中，就可以直接将地址文档嵌入到人员文档中。使用得当的话，内嵌文档会使信息的表示方式更加自然（通常也会更高效）。

MongoDB 这样做的坏处就是会导致更多的数据重复。假设“address”是关系数据库中的一个独立的表，我们需要修正地址中的拼写错误。当我们对“people”和

"address"执行连接操作时，使用这个地址的每个人的信息都会得到更新。但是在 MongoDB 中，则需要对每个人的文档分别修正拼写错误。

2.6.5 _id和ObjectId

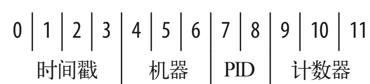
MongoDB 中存储的文档必须有一个 "_id" 键。这个键的值可以是任何类型的，默认是个 ObjectId 对象。在一个集合里面，每个文档都有唯一的 "_id"，确保集合里面每个文档都能被唯一标识。如果有两个集合的话，两个集合可以都有一个 "_id" 的值为 123，但是每个集合里面只能有一个文档的 "_id" 值为 123。

1. ObjectId

ObjectId 是 "_id" 的默认类型。它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。这是 MongoDB 采用 ObjectId，而不是其他比较常规的做法（比如自动增加的主键）的主要原因，因为在多个服务器上同步自动增加主键值既费力又费时。因为设计 MongoDB 的初衷就是用作分布式数据库，所以能够在分片环境中生成唯一的标示符非常重要。

ObjectId 使用 12 字节的存储空间，是一个由 24 个十六进制数字组成的字符串（每个字节可以存储两个十六进制数字）。由于看起来很长，不少人会觉得难以处理。但关键是要知道这个长长的 ObjectId 是实际存储数据的两倍长。

如果快速连续创建多个 ObjectId，会发现每次只有最后几位数字有变化。另外，中间的几位数字也会变化（要是在创建的过程中停顿几秒钟）。这是 ObjectId 的创建方式导致的。ObjectId 的 12 字节按照如下方式生成：



ObjectId 的前 4 个字节是从标准纪元开始的时间戳，单位为秒。这会带来一些有用的属性。

- 时间戳，与随后的 5 字节（稍后介绍）组合起来，提供了秒级别的唯一性。
- 由于时间戳在前，这意味着 ObjectId 大致会按照插入的顺序排列。这对于某些方面很有用，比如可以将其作为索引提高效率，但是这个是没有保证的，仅仅是"大致"。
- 这 4 字节也隐含了文档创建的时间。绝大多数驱动程序都会提供一个方法，用于从 ObjectId 获取这些信息。

因为使用的是当前时间，很多用户担心要对服务器进行时钟同步。虽然在某些情况

下，在服务器间进行时间同步确实是个好主意（参见 23.6.1 节），但是这里其实没有必要，因为时间戳的实际值并不重要，只要它总是不停增加就好了（每秒一次）。

接下来的 3 字节是所在主机的唯一标识符。通常是机器主机名的散列值（hash）。这样就可以确保不同主机生成不同的 `ObjectId`，不产生冲突。

为了确保在同一台机器上并发的多个进程产生的 `ObjectId` 是唯一的，接下来的两字节来自产生 `ObjectId` 的进程的进程标识符（PID）。

前 9 字节保证了同一秒钟不同机器不同进程产生的 `ObjectId` 是唯一的。最后 3 字节是一个自动增加的计数器，确保相同进程同一秒产生的 `ObjectId` 也是不一样的。一秒钟最多允许每个进程拥有 256^3（16 777 216）个不同的 `ObjectId`。

2. 自动生成`_id`

前面讲到，如果插入文档时没有 `"_id"` 键，系统会自动帮你创建一个。可以由 MongoDB 服务器来做这件事，但通常会在客户端由驱动程序完成。这一做法非常好地体现了 MongoDB 的哲学：能交给客户端驱动程序来做的事情就不要交给服务器来做。这种理念背后的原因是，即便是像 MongoDB 这样扩展性非常好的数据库，扩展应用层也要比扩展数据库层容易得多。将工作交由客户端来处理，就减轻了数据库扩展的负担。

2.7 使用MongoDB shell

本节将介绍如何将 shell 作为命令行工具的一部分来使用，如何对 shell 进行定制，以及 shell 的一些高级功能。

在上面的例子中，我们只是连接到了一个本地的 mongod 实例。事实上，可以将 shell 连接到任何 MongoDB 实例（只要你的计算机与 MongoDB 实例所在的计算机能够连通）。在启动 shell 时指定机器名和端口，就可以连接到一台不同的机器（或者端口）：

```
$ mongo some-host:30000/myDB
MongoDB shell version: 2.4.0
connecting to: some-host:30000/myDB
>
```

`db` 现在就指向了 some-host:30000 上的 myDB 数据库。

启动 mongo shell 时不连接到任何 mongod 有时很方便。通过 `--nodb` 参数启动 shell，启动时就不会连接任何数据库：

```
$ mongo --nodb
MongoDB shell version: 2.4.0
>
```

启动之后，在需要时运行 `new Mongo(`*`hostname`*`)` 命令就可以连接到想要的 mongod 了：

```
> conn = new Mongo("some-host:30000")
connection to some-host:30000
> db = conn.getDB("myDB")
myDB
```

执行完这些命令之后，就可以像平常一样使用 `db` 了。任何时候都可以使用这些命令来连接到不同的数据库或者服务器。

2.7.1　shell小贴士

由于 `mongo` 是一个简化的 JavaScript shell，可以通过查看 JavaScript 的在线文档得到大量帮助。对于 MongoDB 特有的功能，shell 内置了帮助文档，可以使用 `help` 命令查看：

```
> help
    db.help()          help on db methods
    db.mycoll.help()   help on collection methods
    sh.help()          sharding helpers
    ...

    show dbs           show database names
    show collections   show collections in current database
    show users         show users in current database
    ...
```

可以通过 `db.help()` 查看数据库级别的帮助，使用 `db.foo.help()` 查看集合级别的帮助。

如果想知道一个函数是做什么用的，可以直接在 shell 输入函数名（函数名后不要输入小括号），这样就可以看到相应函数的 JavaScript 实现代码。例如，如果想知道 `update` 函数的工作机制，或者是记不清参数的顺序，就可以像下面这样做：

```
> db.foo.update
function (query, obj, upsert, multi) {
    assert(query, "need a query");
    assert(obj, "need an object");
    this._validateObject(obj);
    this._mongo.update(this._fullName, query, obj,
                       upsert ? true : false, multi ? true : false);
}
```

2.7.2 使用shell执行脚本

本书其他章都是以交互方式使用 shell，但是也可以将希望执行的 JavaScript 文件传给 shell。直接在命令行中传递脚本就可以了：

```
$ mongo script1.js script2.js script3.js
MongoDB shell version: 2.4.0
connecting to: test
I am script1.js
I am script2.js
I am script3.js
$
```

mongo shell 会依次执行传入的脚本，然后退出。

如果希望使用指定的主机 / 端口上的 mongod 运行脚本，需要先指定地址，然后再跟上脚本文件的名称：

```
$ mongo --quiet server-1:30000/foo script1.js script2.js script3.js
```

这样可以将 `db` 指向 server-1:30000 上的 foo 数据库，然后执行这三个脚本。如上所示，运行 shell 时指定的命令行选项要出现在地址之前。

可以在脚本中使用 `print()` 函数将内容输出到标准输出（stdout），如上面的脚本所示。这样就可以在 shell 中使用管道命令。如果将 shell 脚本的输出管道给另一个使用 `--quiet` 选项的命令，就可以让 shell 不打印“MongoDB shell version…”提示。

也可以使用 `load()` 函数，从交互式 shell 中运行脚本：

```
> load("script1.js")
I am script1.js
>
```

在脚本中可以访问 `db` 变量，以及其他全局变量。然而，shell 辅助函数（比如 `"use db"` 和 `"show collections"`）不可以在文件中使用。这些辅助函数都有对应的 JavaScript 函数，如表 2-1 所示。

表2-1 shell辅助函数对应的JavaScript函数

辅助函数	等价函数
`use foo`	`db.getSisterDB("foo")`
`show dbs`	`db.getMongo().getDBs()`
`show collections`	`db.getCollectionNames()`

可以使用脚本将变量注入到 shell。例如，可以在脚本中简单地初始化一些常用的辅助函数。例如，下面的脚本对于本书的复制和分片部分内容非常有用。这个脚本定

义了一个 connectTo() 函数，它连接到指定端口处的一个本地数据库，并且将 db 指向这个连接。

```
// defineConnectTo.js

/**
 * 连接到指定的数据库，并且将 db 指向这个连接
 */
var connectTo = function(port, dbname) {
    if (!port) {
        port = 27017;
    }

    if (!dbname) {
        dbname = "test";
    }

    db = connect("localhost:"+port+"/"+dbname);
    return db;
};
```

如果在 shell 中加载这个脚本，connectTo 函数就可以使用了。

```
> typeof connectTo
undefined
> load('defineConnectTo.js')
> typeof connectTo
function
```

除了添加辅助函数，还可以使用脚本将通用的任务和管理活动自动化。

默认情况下，shell 会在运行 shell 时所处的目录中查找脚本（可以使用 run("pwd") 命令查看）。如果脚本不在当前目录中，可以为 shell 指定一个相对路径或者绝对路径。例如，如果脚本放置在 ~/my-scripts 目录中，可以使用 load("/home/myUser/my-scripts/defineConnectTo.js") 命令来加载 defineConnectTo.js。注意，load 函数无法解析 ~ 符号。

也可以在 shell 中使用 run() 函数来执行命令行程序。可以在函数参数列表中指定程序所需的参数：

```
> run("ls", "-l", "/home/myUser/my-scripts/")
sh70352| -rw-r--r--  1  myUser myUser 2012-12-13 13:15 defineConnectTo.js
sh70532| -rw-r--r--  1  myUser myUser 2013-02-22 15:10 script1.js
sh70532| -rw-r--r--  1  myUser myUser 2013-02-22 15:12 script2.js
sh70532| -rw-r--r--  1  myUser myUser 2013-02-22 15:13 script3.js
```

通常来说，这种使用方式的局限性非常大，因为输出格式很奇怪，而且不支持管道。

2.7.3 创建.mongorc.js文件

如果某些脚本会被频繁加载，可以将它们添加到 .mongorc.js 文件中。这个文件会在启动 shell 时自动运行。

例如，我们希望启动成功时让 shell 显示一句欢迎语。为此，我们在用户主目录下创建一个名为 .mongorc.js 的文件，向其中添加如下内容：

```
// mongorc.js

var compliment = ["attractive", "intelligent", "like Batman"];
var index = Math.floor(Math.random()*3);

print("Hello, you're looking particularly "+compliment[index]+" today!");
```

然后，当启动 shell 时，就会看到这样一些内容：

```
$ mongo
MongoDB shell version: 2.4.0-pre-
connecting to: test
Hello, you're looking particularly like Batman today!
>
```

为了实用，可以使用这个脚本创建一些自己需要的全局变量，或者是为太长的名字创建一个简短的别名，也可以重写内置的函数。.mongorc.js 最常见的用途之一是移除那些比较“危险”的 shell 辅助函数。可以在这里集中重写这些方法，比如为 `dropDatabase` 或者 `deleteIndexes` 等辅助函数添加 `no` 选项，或者取消它们的定义。

```
var no = function() {
    print("Not on my watch.");
};

// 禁止删除数据库
db.dropDatabase = DB.prototype.dropDatabase = no;

// 禁止删除集合
DBCollection.prototype.drop = no;

// 禁止删除索引
DBCollection.prototype.dropIndex = no;
```

改变数据库函数时，要确保同时对 `db` 变量和 `DB` 原型进行改变（如上例所示）。如果只改变了其中一个，那么 `db` 变量可能没有改变，或者这些改变在新使用的所有数据库（运行 `use anotherDB` 命令）中都不会生效。

现在，如果试图调用这些函数，就会得到一条错误提示。注意，这种方式并不能保护数据库免受恶意用户的攻击，只能预防自己的手误。

如果在启动 shell 时指定 `--norc` 参数，就可以禁止加载 .mongorc.js。

2.7.4 定制shell提示

将 `prompt` 变量设为一个字符串或者函数，就可以重写默认的 shell 提示。例如，如果正在运行一个需要耗时几分钟的查询，你可能希望完成时在 shell 提示中输出当前时间，这样就可以知道最后一个操作的完成时间了。

```
prompt = function() {
    return (new Date())+"> ";
};
```

另一个方便的提示是显示当前使用的数据库：

```
prompt = function() {
    if (typeof db == 'undefined') {
        return '(nodb)> ';
    }

    // 检查最后的数据库操作
    try {
        db.runCommand({getLastError:1});
    }
    catch (e) {
        print(e);
    }

    return db+"> ";
};
```

注意，提示函数应该返回字符串，而且应该小心谨慎地处理异常：如果提示中出现了异常会对用户造成困惑！

通常来说，提示函数中应该包含对 `getLastError` 的调用。这样可以捕获数据库错误，而且可以在 shell 断开时自动重新连接（比如重启了 mongod）。

可以在 .mongorc.js 中定制自己想要的提示。也可以定制多个提示，在 shell 中可以自由切换。

2.7.5 编辑复合变量

shell 的多行支持是非常有限的：不可以编辑之前的行。如果编辑到第 15 行时发现第 1 行有个错误，那会让人非常懊恼。因此，对于大块的代码或者是对象，你可能更愿意在编辑器中编辑。为了方便地调用编辑器，可以在 shell 中设置 `EDITOR` 变量（也可以在环境变量中设置）：

```
> EDITOR="/usr/bin/emacs"
```

现在，如果想要编辑一个变量，可以使用 "edit 变量名 " 这个命令，比如：

```
> var wap = db.books.findOne({title: "War and Peace"})
> edit wap
```

修改完成之后，保存并退出编辑器。变量就会被重新解析然后加载回 shell。

在 .mongorc.js 文件中添加一行内容，`EDITOR="` 编辑器路径 `";`，以后就不必单独设置 `EDITOR` 变量了。

2.7.6 集合命名注意事项

可以使用 `db.`*`collectionName`* 获取一个集合的内容，但是，如果集合名称中包含保留字或者无效的 JavaScript 属性名称，`db.`*`collectionName`* 就不能正常工作了。

假设要访问 version 集合，不能直接使用 `db.version`，因为 `db.version` 是 `db` 的一个方法（会返回当前 MongoDB 服务器的版本）：

```
> db.version
function () {
    return this.serverBuildInfo().version;
}
```

为了访问 version 集合，必须使用 `getCollection` 函数：

```
> db.getCollection("version");
test.version
```

如果集合名称中包含无效的 JavaScript 属性名称（比如 foo-bar-baz 和 123abc），也可以使用这个函数来访问相应的集合。（注意，JavaScript 属性名称只能包含字母、数字，以及 `"$"` 和 `"_"` 字符，而且不能以数字开头。）

还有一种方法可以访问以无效属性名称命名的集合，那就是使用数组访问语法：在 JavaScript 中，`x.y` 等同于 `x['y']`。也就是说，除了名称的字面量之外，还可以使用变量访问子集合。因此，如果需要对 blog 的每一个子集合进行操作，可以使用如下方式进行迭代：

```
var collections = ["posts", "comments", "authors"];
for (var i in collections) {
    print(db.blog[collections[i]]);
}
```

而不必这样：

```
print(db.blog.posts);
print(db.blog.comments);
print(db.blog.authors);
```

注意，不能使用 db.blog.i，这样会被解释为 test.blog.i，而不是 test.blog.posts。必须使用 `db.blog[i]` 语法才能将 `i` 解释为相应的变量。

可以使用这种方式来访问那些名字怪异的集合：

```
> var name = "@#&!"
> db[name].find()
```

直接使用 `db.@#&!` 进行查询是非法的，但是可以使用 `db[name]`。

第3章

创建、更新和删除文档

本章会介绍对数据库移入 / 移出数据的基本操作，具体包含如下操作：

- 向集合添加新文档；
- 从集合里删除文档；
- 更新现有文档；
- 为这些操作选择合适的安全级别和速度。

3.1 插入并保存文档

插入是向 MongoDB 中添加数据的基本方法。可以使用 `insert` 方法向目标集合插入一个文档：

```
> db.foo.insert({"bar" : "baz"})
```

这个操作会给文档自动增加一个 `"_id"` 键（要是原来没有的话），然后将其保存到 MongoDB 中。

3.1.1 批量插入

如果要向集合中插入多个文档，使用批量插入会快一些。使用批量插入，可以将一组文档传递给数据库。

在 shell 中，可以使用 `batchInsert` 函数实现批量插入，它与 `insert` 函数非常像，

只是它接受的是一个文档数组作为参数：

```
> db.foo.batchInsert([{"_id" : 0}, {"_id" : 1}, {"_id" : 2}])
> db.foo.find()
{ "_id" : 0 }
{ "_id" : 1 }
{ "_id" : 2 }
```

一次发送数十、数百乃至数千个文档会明显提高插入的速度。

只有需要将多个文档插入到一个集合时，这种方式才会有用。不能在单次请求中将多个文档批量插入到多个集合中。要是只导入原始数据（例如，从数据 feed 或者 MySQL 中导入），可以使用命令行工具，如 `mongoimport`，而不是批量插入。另一方面，可以使用批量插入在将数据存入 MongoDB 之前对数据做一些小的修整（将日期转换为日期类型，或添加自定义的 `"_id"`），这样批量插入也可用于导入数据。

当前版本的 MongoDB 能接受的最大消息长度是 48 MB，所以在一次批量插入中能插入的文档是有限制的。如果试图插入 48 MB 以上的数据，多数驱动程序会将这个批量插入请求拆分为多个 48 MB 的批量插入请求。具体可以查看所使用的驱动程序的相关文档。

如果在执行批量插入的过程中有一个文档插入失败，那么在这个文档之前的所有文档都会成功插入到集合中，而这个文档以及之后的所有文档全部插入失败。

```
> db.foo.batchInsert([{"_id" : 0}, {"_id" : 1}, {"_id" : 1}, {"_id" : 2}])
```

只有前两个文档会被插入，因为插入第三个文档时会发生错误：集合中已经存在一个 `_id` 为 1 的文档，不能重复插入。

在批量插入中遇到错误时，如果希望 `batchInsert` 忽略错误并且继续执行后续插入，可以使用 `continueOnError` 选项。这样就可以将上面例子中的第一个、第二个以及第四个文档都插入到集合中。shell 并不支持这个选项，但是所有驱动程序都支持。

3.1.2 插入校验

插入数据时，MongoDB 只对数据进行最基本的检查：检查文档的基本结构，如果没有 `"_id"` 字段，就自动增加一个。检查大小就是其中一项基本结构检查：所有文档都必须小于 16 MB（这个值是 MongoDB 设计者人为定的，未来有可能会增加）。作这样的限制主要是为了防止不良的模式设计，并且保证性能一致。如果要查看 *doc* 文档的 BSON 大小（单位为字节），可以在 shell 中执行 `Object.bsonsize(`*doc*`)`。

16 MB 的数据究竟有多大？要知道整部《战争与和平》也才 3.14 MB。

由于 MongoDB 只进行最基本的检查，所以插入非法数据很容易（如果你想这么干的话）。因此，应该只允许信任的源（比如你的应用程序服务器）连接数据库。主流语言的所有驱动程序（以及大部分其他语言的驱动程序），都会在将数据插入到数据库之前做大量的数据校验（比如文档是否过大，文档是否包含非 UTF-8 字符串，是否使用不可识别的类型）。

3.2 删除文档

现在数据库中有些数据，要删除它：

```
> db.foo.remove()
```

上述命令会删除 foo 集合中的所有文档。但是不会删除集合本身，也不会删除集合的元信息。

`remove` 函数可以接受一个查询文档作为可选参数。给定这个参数以后，只有符合条件的文档才被删除。例如，假设要删除 mailing.list 集合中所有 `"opt-out"` 为 `true` 的人：

```
> db.mailing.list.remove({"opt-out" : true})
```

删除数据是永久性的，不能撤销，也不能恢复。

删除速度

删除文档通常很快，但是如果要清空整个集合，那么使用 `drop` 直接删除集合会更快（然后在这个空集合上重建各项索引）。

例如，使用如下方法插入一百万个测试数据：

```
> for (var i = 0; i < 1000000; i++) {
... db.tester.insert({"foo": "bar", "baz": i, "z": 10 - i})
... }
```

现在把刚插入的文档都删除，并记录花费的时间。首先使用 `remove` 进行删除：

```
> var timeRemoves = function() {
... var start = (new Date()).getTime();
...
... db.tester.remove();
... db.findOne(); // makes sure the remove finishes before continuing
...
```

```
... var timeDiff = (new Date()).getTime() - start;
... print("Remove took: "+timeDiff+"ms");
... }
> timeRemoves()
```

在 MacBookAir 笔记本电脑上，这段脚本输出“Removetook:9676ms”。

如果用 `db.tester.drop()` 代替 `remove` 和 `findOne`，只用 1 ms！速度提升相当明显，但也是有代价的：不能指定任何限定条件。整个集合都被删除了，所有元数据也都不见了。

3.3 更新文档

文档存入数据库以后，就可以使用 `update` 方法来更新它。`update` 有两个参数，一个是查询文档，用于定位需要更新的目标文档；另一个是修改器（modifier）文档，用于说明要对找到的文档进行哪些修改。

更新操作是不可分割的：若是两个更新同时发生，先到达服务器的先执行，接着执行另外一个。所以，两个需要同时进行的更新会迅速接连完成，此过程不会破坏文档：最新的更新会取得“胜利”。

3.3.1 文档替换

最简单的更新就是用一个新文档完全替换匹配的文档。这适用于进行大规模模式迁移的情况。例如，要对下面的用户文档做一个比较大的调整：

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "name" : "joe",
    "friends" : 32,
    "enemies" : 2
}
```

我们希望将 `"friends"` 和 `"enemies"` 两个字段移到 `"relationships"` 子文档中。可以在 shell 中改变文档的结构，然后使用 `update` 替换数据库中的当前文档：

```
> var joe = db.users.findOne({"name" : "joe"});
> joe.relationships = {"friends" : joe.friends, "enemies" : joe.enemies};
{
    "friends" : 32,
    "enemies" : 2
}> joe.username = joe.name;
"joe"
> delete joe.friends;
true
```

```
> delete joe.enemies;
true
> delete joe.name;
true
> db.users.update({"name" : "joe"}, joe);
```

现在，用 findOne 查看更新后的文档结构。

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "username" : "joe",
    "relationships" : {
        "friends" : 32,
        "enemies" : 2
    }
}
```

一个常见的错误是查询条件匹配到了多个文档，然后更新时由于第二个参数的存在就产生重复的 "_id" 值。数据库会抛出错误，任何文档都不会更新。

例如，有好几个文档都有相同的 "name" 值，但是我们没有意识到：

```
> db.people.Ṅnd()
{"_id" : ObjectId("4b2b9f67a1f631733d917a7b"), "name" : "joe", "age" : 65},
{"_id" : ObjectId("4b2b9f67a1f631733d917a7c"), "name" : "joe", "age" : 20},
{"_id" : ObjectId("4b2b9f67a1f631733d917a7d"), "name" : "joe", "age" : 49},
```

现在如果第二个 Joe 过生日，要增加 "age" 的值，我们可能会这么做：

```
> joe = db.people.findOne({"name" : "joe", "age" : 20});
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7c"),
    "name" : "joe",
    "age" : 20
}
> joe.age++;
> db.people.update({"name" : "joe"}, joe);
E11001 duplicate key on update
```

到底怎么了？调用 update 时，数据库会查找一个 "name" 值为 "Joe" 的文档。找到的第一个是 65 岁的 Joe。然后数据库试着用变量 joe 中的内容替换找到的文档，但是会发现集合里面已经有一个具有同样 "_id" 的文档。所以，更新就会失败，因为 "_id" 值必须唯一。为了避免这种情况，最好确保更新时总是指定一个唯一文档，例如使用 "_id" 这样的键来匹配。对于上面的例子，这才是正确的更新方法：

```
> db.people.update({"_id" : ObjectId("4b2b9f67a1f631733d917a7c")}, joe)
```

使用 "_id" 作为查询条件比使用随机字段速度更快，因为是通过 "_id" 建立的索引。第 5 章会介绍索引对更新和其他操作的影响。

3.3.2 使用修改器

通常文档只会有一部分要更新。可以使用原子性的*更新修改器*（update modifier），指定对文档中的某些字段进行更新。更新修改器是种特殊的键，用来指定复杂的更新操作，比如修改、增加或者删除键，还可能是操作数组或者内嵌文档。

假设要在一个集合中放置网站的分析数据，只要有人访问页面，就增加计数器。可以使用更新修改器原子性地完成这个增加。每个 URL 及对应的访问次数都以如下方式存储在文档中：

```
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 52
}
```

每次有人访问页面，就通过 URL 找到该页面，并用 `"$inc"` 修改器增加 `"pageviews"` 的值。

```
> db.analytics.update({"url" : "www.example.com"},
... {"$inc" : {"pageviews" : 1}})
```

现在，执行一个 `find` 操作，会发现 `"pageviews"` 的值增加了 1。

```
> db.analytics.find()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 53
}
```

使用修改器时，`"_id"` 的值不能改变。（注意，整个文档替换时可以改变 `"_id"`。）其他键值，包括其他唯一索引的键，都是可以更改的。

1. "$set"修改器入门

`"$set"` 用来指定一个字段的值。如果这个字段不存在，则创建它。这对更新模式或者增加用户定义的键来说非常方便。例如，用户资料存储在下面这样的文档里：

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin"
}
```

非常简要的一段用户信息。要想添加喜欢的书籍进去，可以使用 "$set"：

```
> db.users.update({"_id" : ObjectId("4b253b067525f35f94b60a31")},
... {"$set" : {"favorite book" : "War and Peace"}})
```

之后文档就有了 "favorite book" 键。

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin",
    "favorite book" : "War and Peace"
}
```

要是用户觉得喜欢的其实是另外一本书，"$set" 又能帮上忙了：

```
> db.users.update({"name" : "joe"},
... {"$set" : {"favorite book" : "Green Eggs and Ham"}})
```

用 "$set" 甚至可以修改值的类型。例如，如果用户觉得喜欢很多本书，就可以将 "favorite book" 键的值变成一个数组：

```
> db.users.update({"name" : "joe"},
... {"$set" : {"favorite book" :
...     ["Cat's Cradle", "Foundation Trilogy", "Ender's Game"]}})
```

如果用户突然发现自己其实不爱读书，可以用 "$unset" 将这个键完全删除：

```
> db.users.update({"name" : "joe"},
... {"$unset" : {"favorite book" : 1}})
```

现在这个文档就和刚开始时一样了。

也可以用 "$set" 修改内嵌文档：

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe",
        "email" : "joe@example.com"
    }
}
> db.blog.posts.update({"author.name" : "joe"},
... {"$set" : {"author.name" : "joe schmoe"}})
> db.blog.posts.findOne()
```

```
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe schmoe",
        "email" : "joe@example.com"
    }
}
```

增加、修改或删除键时，应该使用 $ 修改器。要把 "foo" 的值设为 "bar"，常见的错误做法如下：

```
> db.coll.update(criteria, {"foo" : "bar"})
```

这会事与愿违。实际上这会将整个文档用 {"foo":"bar"} 替换掉。一定要使用以 $ 开头的修改器来修改键 / 值对。

2. 增加和减少

"$inc" 修改器用来增加已有键的值，或者该键不存在那就创建一个。对于更新分析数据、因果关系、投票或者其他有变化数值的地方，使用这个都会非常方便。

假如建立了一个游戏集合，将游戏和变化的分数都存储在里面。比如用户玩弹球（pinball）游戏，可以插入一个包含游戏名和玩家的文档来标识不同的游戏：

```
> db.games.insert({"game" : "pinball", "user" : "joe"})
```

要是小球撞到了砖块，就会给玩家加分。分数可以随便给，这里就把玩家得分基数约定成 50 好了。使用 "$inc" 修改器给玩家加 50 分：

```
> db.games.update({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 50}})
```

更新后，可以看到：

```
> db.games.findOne()
{
     "_id" : ObjectId("4b2d75476cc613d5ee930164"),
     "game" : "pinball",
     "user" : "joe",
     "score" : 50
 }
```

分数（score）键原来并不存在，所以 "$inc" 创建了这个键，并把值设定成增加量：50。

如果小球落入加分区，要加 10 000 分。只要给 "$inc" 传递一个不同的值就好了：

```
> db.games.update({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 10000}})
```

现在来看看结果：

```
> db.games.find()
{
     "_id" : ObjectId("4b2d75476cc613d5ee930164"),
     "game" : "pinball",
     "user" : "joe",
     "score" : 10050
}
```

`"score"` 键已经有了，而且有一个数字类型的值，所以服务器就给这个值增加了 10 000。

`"$inc"` 与 `"$set"` 的用法类似，就是专门来增加（和减少）数字的。`"$inc"` 只能用于整型、长整型或双精度浮点型的值。要是用在其他类型的数据上就会导致操作失败，例如 `null`、布尔类型以及数字构成的字符串，而在其他很多语言中，这些类型都会自动转换为数值类型。

```
> db.foo.insert({"count" : "1"})
> db.foo.update({}, {"$inc" : {"count" : 1}})
Cannot apply $inc modifier to non-number
```

另外，`"$inc"` 键的值必须为数字。不能使用字符串、数组或其他非数字的值。否则就会提示“Modifier `"$inc"` allowed for numbers only”（修改器 `"$inc"` 只允许使用数值类型）这样的错误。要修改其他类型，应该使用 `"$set"` 或者一会儿要讲到的数组修改器。

3. 数组修改器

有一大类很重要的修改器可用于操作数组。数组是常用且非常有用的数据结构：它们不仅是可通过索引进行引用的列表，而且还可以作为数据集（set）来用。

4. 添加元素

如果数组已经存在，`"$push"` 会向已有的数组末尾加入一个元素，要是没有就创建一个新的数组。例如，假设要存储博客文章，要添加一个用于保存数组的 `"comments"`（评论）键。可以向还不存在的 `"comments"` 数组添加一条评论，这个数组会被自动创建，并加入一条评论：

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "..."
}
```

```
> db.blog.posts.update({"title" : "A blog post"},
... {"$push" : {"comments" :
...     {"name" : "joe", "email" : "joe@example.com",
...     "content" : "nice post."}}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        }
    ]
}
```

要是还想添加一条评论，继续使用 "$push"：

```
> db.blog.posts.update({"title" : "A blog post"},
... {"$push" : {"comments" :
...     {"name" : "bob", "email" : "bob@example.com",
...     "content" : "good post."}}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

这是一种比较简单的 "$push" 使用形式，也可以将它应用在一些比较复杂的数组操作中。使用 "$each" 子操作符，可以通过一次 "$push" 操作添加多个值。

```
> db.stock.ticker.update({"_id" : "GOOG"},
... {"$push" : {"hourly" : {"$each" : [562.776, 562.790, 559.123]}}})
```

这样就可以将三个新元素添加到数组中。如果指定的数组中只含有一个元素，那这个操作就等同于没有使用 "$each" 的普通 "$push" 操作。

如果希望数组的最大长度是固定的，那么可以将 "$slice" 和 "$push" 组合在一起使用，这样就可以保证数组不会超出设定好的最大长度，这实际上就得到了一个最多包含 N 个元素的数组：

```
> db.movies.update({"genre" : "horror"},
... {"$push" : {"top10" : {
...     "$each" : ["Nightmare on Elm Street", "Saw"],
...     "$slice" : -10}}})
```

这个例子会限制数组只包含最后加入的 10 个元素。"$slice" 的值必须是负整数。

如果数组的元素数量小于 10（"$push" 之后），那么所有元素都会保留。如果数组的元素数量大于 10，那么只有最后 10 个元素会保留。因此，"$slice" 可以用来在文档中创建一个队列。

最后，可以在清理元素之前使用 "$sort"，只要向数组中添加子对象就需要清理：

```
> db.movies.find({"genre" : "horror"},
... {"$push" : {"top10" : {
...     "$each" : [{"name" : "Nightmare on Elm Street", "rating" : 6.6},
...                {"name" : "Saw", "rating" : 4.3}],
...     "$slice" : -10,
...     "$sort" : {"rating" : -1}}}})
```

这样会根据 "rating" 字段的值对数组中的所有对象进行排序，然后保留前 10 个。注意，不能只将 "$slice" 或者 "$sort" 与 "$push" 配合使用，且必须使用 "$each"。

5. 将数组作为数据集使用

你可能想将数组作为数据集使用，保证数组内的元素不会重复。可以在查询文档中用 "$ne" 来实现。例如，要是作者不在引文列表中，就添加进去，可以这么做：

```
> db.papers.update({"authors cited" : {"$ne" : "Richie"}},
... {$push : {"authors cited" : "Richie"}})
```

也可以用 "$addToSet" 来实现，要知道有些情况 "$ne" 根本行不通，有些时候更适合用 "$addToSet"。

例如，有一个表示用户的文档，已经有了电子邮件地址的数据集：

```
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com"
```

```
        ]
    }
```

添加新地址时，用 "$addToSet" 可以避免插入重复地址：

```
    > db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
    ... {"$addToSet" : {"emails" : "joe@gmail.com"}})
    > db.users.ǸndOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
    {
        "_id" : ObjectId("4b2d75476cc613d5ee930164"),
        "username" : "joe",
        "emails" : [
            "joe@example.com",
            "joe@gmail.com",
            "joe@yahoo.com",
        ]
    }
    > db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
    ... {"$addToSet" : {"emails" : "joe@hotmail.com"}})
    > db.users.ǸndOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
    {
        "_id" : ObjectId("4b2d75476cc613d5ee930164"),
        "username" : "joe",
        "emails" : [
            "joe@example.com",
            "joe@gmail.com",
            "joe@yahoo.com",
            "joe@hotmail.com"
        ]
    }
```

将 "$addToSet" 和 "$each" 组合起来，可以添加多个不同的值，而用 "$ne" 和 "$push" 组合就不能实现。例如，想一次添加多个邮件地址，就可以使用这些修改器：

```
    > db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")}, {"$addToSet" :
    ... {"emails" : {"$each" :
    ...     ["joe@php.net", "joe@example.com", "joe@python.org"]}}})
    > db.users.ǸndOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
    {
        "_id" : ObjectId("4b2d75476cc613d5ee930164"),
        "username" : "joe",
        "emails" : [
            "joe@example.com",
            "joe@gmail.com",
            "joe@yahoo.com",
            "joe@hotmail.com"
            "joe@php.net"
            "joe@python.org"
        ]
    }
```

6. 删除元素

有几个从数组中删除元素的方法。若是把数组看成队列或者栈，可以用 "$pop"，这个修改器可以从数组任何一端删除元素。{"$pop":{"*key*":1}} 从数组末尾删除一个元素，{"$pop":{"*key*":-1}} 则从头部删除。

有时需要基于特定条件来删除元素，而不仅仅是依据元素位置，这时可以使用 "$pull"。例如，有一个无序的待完成事项列表：

```
> db.lists.insert({"todo" : ["dishes", "laundry", "dry cleaning"]})
```

如果我们先完成了洗衣服（laundry），就可以用下面的方式删除它：

```
> db.lists.update({}, {"$pull" : {"todo" : "laundry"}})
```

通过查找，会发现只有两个元素了：

```
> db.lists.find()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "todo" : [
        "dishes",
        "dry cleaning"
    ]
}
```

"$pull" 会将所有匹配的文档删除，而不是只删除一个。对数组 [1,1,2,1] 执行 pull 1，结果得到 只有一个元素的数组 [2]。

数组操作符只能用于包含数组值的键。例如，不能将一个整数插入数组，也不能将一个字符串从数组中弹出。要修改标量值，使用 "$set" 或者 "$inc"。

7. 基于位置的数组修改器

若是数组有多个值，而我们只想对其中的一部分进行操作，就需要一些技巧。有两种方法操作数组中的值：通过位置或者定位操作符（"$"）。

数组下标都是以 0 开头的，可以将下标直接作为键来选择元素。例如，这里有个文档，其中包含由内嵌文档组成的数组，比如包含评论的博客文章。

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b329a216cc613d5ee930192"),
    "content" : "...",
    "comments" : [
        {
            "comment" : "good post",
            "author" : "John",
```

```
                "votes" : 0
            },
            {
                "comment" : "i thought it was too short",
                "author" : "Claire",
                "votes" : 3
            },
            {
                "comment" : "free watches",
                "author" : "Alice",
                "votes" : -1
            }
        ]
    }
```

如果想增加第一个评论的投票数量，可以这么做：

```
> db.blog.update({"post" : post_id},
... {"$inc" : {"comments.0.votes" : 1}})
```

但是很多情况下，不预先查询文档就不能知道要修改的数组的下标。为了克服这个困难，MongoDB 提供了定位操作符 "$"，用来定位查询文档已经匹配的数组元素，并进行更新。例如，要是用户 John 把名字改成了 Jim，就可以用定位符替换他在评论中的名字：

```
db.blog.update({"comments.author" : "John"},
... {"$set" : {"comments.$.author" : "Jim"}})
```

定位符只更新第一个匹配的元素。所以，如果 John 发表了多条评论，那么他的名字只在第一条评论中改变。

8. 修改器速度

有的修改器运行比较快。$inc 能*就地*修改，因为不需要改变文档的大小，只需要将键的值修改一下（对文档大小的改变非常小），所以非常快。而数组修改器可能会改变文档的大小，就会慢一些（"$set" 能在文档大小不发生变化时立即修改它，否则性能也会有所下降）。

将文档插入到 MongoDB 中时，依次插入的文档在磁盘上的位置是相邻的。因此，如果一个文档变大了，原先的位置就放不下这个文档了，这个文档就会被移动到集合中的另一个位置。

可以在实际操作中看到这种变化。创建一个包含几个文档的集合，对某个位于中间的文档进行修改，使其尺寸变大。然后会发现这个文档被移动到了集合的尾部：

```
> db.coll.insert({"x" :"a"})
> db.coll.insert({"x" :"b"})
```

```
> db.coll.insert({"x" :"c"})
> db.coll.find()
{ "_id" : ObjectId("507c3581d87d6a342e1c81d3"), "x" : "a" }
{ "_id" : ObjectId("507c3583d87d6a342e1c81d4"), "x" : "b" }
{ "_id" : ObjectId("507c3585d87d6a342e1c81d5"), "x" : "c" }
> db.coll.update({"x" : "b"}, {$set: {"x" : "bbb"}})
> db.coll.find()
{ "_id" : ObjectId("507c3581d87d6a342e1c81d3"), "x" : "a" }
{ "_id" : ObjectId("507c3585d87d6a342e1c81d5"), "x" : "c" }
{ "_id" : ObjectId("507c3583d87d6a342e1c81d4"), "x" : "bbb" }
```

MongoDB 不得不移动一个文档时，它会修改集合的*填充因子*（padding factor）。填充因子是 MongoDB 为每个新文档预留的增长空间。可以运行 `db.coll.stats()` 查看填充因子。执行上面的更新之前，`"paddingFactor"` 字段的值是 1：根据实际的文档大小，为每个新文档分配精确的空间，不预留任何增长空间，如图 3-1 所示。让其中一个文档增大之后，再次运行这个命令（如图 3-2 所示），会发现填充因子增加到了 1.5：为每个新文档预留其一半大小的空间作为增长空间，如图 3-2 所示。如果随后的更新导致了更多次的文档移动，填充因子会持续变大（虽然不会像第一次移动时的变化那么大）。如果不再有文档移动，填充因子的值会缓慢降低，如图 3-3 所示。

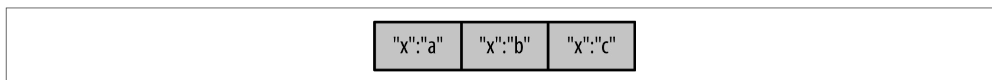


图 3-1：最初文档之间没有多余的空间

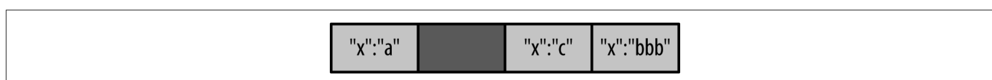


图 3-2：如果一个文档因为体积变大而不得不进行移动，它原先占用的空间就闲置了，而且填充因子会增加

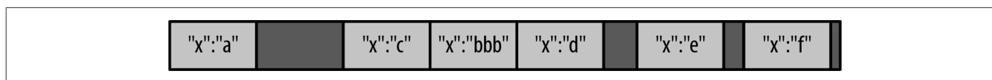


图 3-3：之后插入的新文档都会拥有填充因子指定大小的增长空间。如果在之后的插入中不再发生文档移动，填充因子会逐渐变小

移动文档是非常慢的。MongoDB 必须将文档原先所占的空间释放掉，然后将文档写入另一片空间。因此，应该尽量让填充因子的值接近 1。无法手动设定填充因子的值（除非是要对集合进行压缩，参见 18.4 节），但是可以设计一种不依赖于文档、可以任意增长的模式。第 8 章会详细介绍模式设计的相关内容。

下面用一个简单的程序来展示原地更新和文档移动的速度差别。下面的程序插入了一个只包含一个键的文档，并且对这个键的值进行了 100 000 次增加：

```
> db.tester.insert({"x" : 1})
> var timeInc = function() {
... var start = (new Date()).getTime();
...
... for (var i=0; i<100000; i++) {
...     db.tester.update({}, {"$inc" : {"x" : 1}});
...     db.getLastError();
... }
...
... var timeDiff = (new Date()).getTime() - start;
... print("Updates took: "+timeDiff+"ms");
... }
> timeInc()
```

在 MacBook Air 上，总共花费了 7.33 秒。也就是每秒超过 13 000 次更新。现在，使用 "$push" 向一个只有一个键的数组中插入新数据，重复 100 000 次。将上面例子中用于更新文档的代码修改为：

```
...   db.tester.update({}, {"$push" : {"x" : 1}})
```

这个程序运行时间为 67.58 秒，每秒少于 1500 次更新。

使用 "$push" 以及其他一些数组修改器是非常好的，而且通常是必要的，但是，在进行类似的更新时，需要好好权衡一下。如果 "$push" 成为了瓶颈，那么将一个内嵌文档取出放入一个单独的集合中，手动填充，或者使用第 8 章将要介绍的其他某项技术，都很值得。

写作本书时，MongoDB 仍然不能很好地重用空白空间，因此频繁移动文档会产生大量空的数据文件。如果有太多不能重用的空白空间，你会经常在日志中看到如下信息：

```
Thu Apr  5 01:12:28 [conn124727] info DFM::ǸndAll(): extent a:7f18dc00 was
    empty, skipping ahead
```

这就是说，执行查询时，MongoDB 会在整个范围（entire extent，可以在附录 B 中查看相关定义。简单来说，它就是集合的一个子集）内进行查找，却找不到任何文档：这只是个空白空间。这个消息提示本身没什么影响，但是它指出你当前拥有太多的碎片，可能需要进行压缩。

如果你的模式在进行插入和删除时会进行大量的移动或者是经常打乱数据，可以使用 usePowerOf2Sizes 选项以提高磁盘复用率。可以通过 collMod 命令来设定这个选项：

```
> db.runCommand({"collMod" : collectionName, "usePowerOf2Sizes" : true})
```

这个集合之后进行的所有空间分配，得到的块大小都是 2 的幂。由于这个选项会导致初始空间分配不再那么高效，所以应该只在需要经常打乱数据的集合上使用。在一个只进行插入或者原地更新的集合上使用这个选项，会导致写入速度变慢。

如果在这个命令中指定 `"usePowerOf2Sizes"` 选项的值为 `false`，就会关闭这种特殊分配机制。这个选项只会影响之后新分配的记录，因此，在已有的集合上运行这个命令或者是更改这个选项的值，不会对现有数据产生影响。

3.3.3 upsert

`upsert` 是一种特殊的更新。要是没有找到符合更新条件的文档，就会以这个条件和更新文档为基础创建一个新的文档。如果找到了匹配的文档，则正常更新。`upsert` 非常方便，不必预置集合，同一套代码既可以用于创建文档又可以用于更新文档。

我们回过头看看那个记录网站页面访问次数的例子。要是没有 `upsert`，就得试着查询 URL，没有找到就得新建一个文档，找到的话就增加访问次数。要是把这个写成 JavaScript 程序，会是下面这样的：

```
// 检查这个页面是否有一个文档
blog = db.analytics.findOne({"url" : "/blog"})

// 如果有，就将视图数加 1 并保存
if (blog) {
  blog.pageviews++;
  db.analytics.save(blog);
}
// 否则为这个页面创建一个新文档
else {
  db.analytics.save({"url" : "/blog", pageviews : 1})
}
```

这就是说如果有人访问页面，我们得先对数据库进行查询，然后选择更新或者插入。要是多个进程同时运行这段代码，还会遇到同时对给定 URL 插入多个文档这样的竞态条件。

要是使用 `upsert`，既可以避免竞态问题，又可以缩减代码量（`update` 的第 3 个参数表示这是个 `upsert`）：

```
db.analytics.update({"url" : "/blog"}, {"$inc" : {"pageviews" : 1}}, true)
```

这行代码和之前的代码作用完全一样，但它更高效，并且是原子性的！创建新文档会将条件文档作为基础，然后对它应用修改器文档。

例如，要是执行一个匹配键并增加对应键值的 upsert 操作，会在匹配的文档上进行增加：

```
> db.users.update({"rep" : 25}, {"$inc" : {"rep" : 3}}, true)
> db.users.findOne()
{
    "_id" : ObjectId("4b3295f26cc613d5ee93018f"),
    "rep" : 28
}
```

upsert 创建一个 "rep" 值为 25 的文档，随后将这个值加 3，最后得到 "rep" 为 28 的文档。要是不指定 upsert 选项，{"rep" : 25} 不会匹配任何文档，也就不会对集合进行任何更新。

要是再次运行这个 upsert（条件为 {"rep" : 25}），还会创建一个新文档。这是因为没有文档满足匹配条件（唯一一个文档的 "rep" 值是 28）。

有时，需要在创建文档的同时创建字段并为它赋值，但是在之后的所有更新操作中，这个字段的值都不再改变。这就是 "$setOnInsert" 的作用。"$setOnInsert" 只会在文档插入时设置字段的值。因此，实际使用中可以这么做：

```
> db.users.update({}, {"$setOnInsert" : {"createdAt" : new Date()}}, true)
> db.users.findOne()
{
    "_id" : ObjectId("512b8aefae74c67969e404ca"),
    "createdAt" : ISODate("2013-02-25T16:01:50.742Z")
}
```

如果再次运行这个更新，会匹配到这个已存在的文档，所以不会再插入文档，因此 "createdAt" 字段的值也不会改变：

```
> db.users.update({}, {"$setOnInsert" : {"createdAt" : new Date()}}, true)
> db.users.findOne()
{
    "_id" : ObjectId("512b8aefae74c67969e404ca"),
    "createdAt" : ISODate("2013-02-25T16:01:50.742Z")
}
```

注意，通常不需要保留 "createdAt" 这样的字段，因为 ObjectIds 里包含了一个用于标明文档创建时间的时间戳。但是，在预置或者初始化计数器时，或者是对于不使用 ObjectIds 的集合来说，"$setOnInsert" 是非常有用的。

save shell帮助程序

save 是一个 shell 函数，如果文档不存在，它会自动创建文档；如果文档存在，它就更新这个文档。它只有一个参数：文档。要是这个文档含有 "_id" 键，save 会

调用 upsert。否则，会调用 insert。如果在 Shell 中使用这个函数，就可以非常方便地对文档进行快速修改。

```
> var x = db.foo.findOne()
> x.num = 42
42
> db.foo.save(x)
```

要是不用 save 的话，最后一行代码看起来就会比较繁琐了，比如 db.foo.update({"_id" : x._id}, x)。

3.3.4 更新多个文档

默认情况下，更新只能对符合匹配条件的第一个文档执行操作。要是有多个文档符合条件，只有第一个文档会被更新，其他文档不会发生变化。要更新所有匹配的文档，可以将 update 的第 4 个参数设置为 true。



update 的行为以后可能会发生变化（服务器可能默认会更新所有匹配的文档，只有第 4 个参数为 false 才会只更新一个），所以建议每次都显式表明要不要做多文档更新。

这样不但更明确地指定了 update 的行为，而且可以在默认行为发生变化时正常运行。

多文档更新对模式迁移非常有用，还可以在对特定用户发布新功能时使用。例如，要送给在某个指定日期过生日的所有用户一份礼物，就可以使用多文档更新，将 "gift" 增加到他们的账号：

```
> db.users.update({"birthday" : "10/13/1978"},
... {"$set" : {"gift" : "Happy Birthday!"}}, false, true)
```

这样就给生日为 1978 年 10 月 13 日的所有用户文档添加了 "gift" 键。

想要知道多文档更新到底更新了多少文档，可以运行 getLastError 命令（可以理解为“返回最后一次操作的相关信息”）。键 "n" 的值就是被更新文档的数量。

```
> db.count.update({"x" : 1}, {"$inc" : {"x" : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

这里 "n" 为 5，说明有 5 个文档被更新了。"updatedExisting" 为 true，说明是对已有的文档进行更新。

3.3.5 返回被更新的文档

调用 getLastError 仅能获得关于更新的有限信息，并不能返回被更新的文档。可以通过 findAndModify 命令得到被更新的文档。这对于操作队列以及执行其他需要进行原子性取值和赋值的操作来说，十分方便。

假设我们有一个集合，其中包含以一定顺序运行的进程。其中每个进程都用如下形式的文档表示：

```
{
    "_id" : ObjectId(),
    "status" : state,
    "priority" : N
}
```

"status" 是一个字符串，它的值可以是 "READY"、"RUNNING" 或 "DONE"。需要找到状态为 "READY" 具有最高优先级的任务，运行相应的进程函数，然后将其状态更新为 "DONE"。也可能需要查询已经就绪的进程，按照优先级排序，然后将优先级最高的进程的状态更新为 "RUNNING"。完成了以后，就把状态改为 "DONE"。就像下面这样：

```
var cursor = db.processes.find({"status" : "READY"});
ps = cursor.sort({"priority" : -1}).limit(1).next();
db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "RUNNING"}});
do_something(ps);
db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}});
```

这个算法不是很好，可能会导致竞态条件。假设有两个线程正在运行。A 线程读取了文档，B 线程在 A 将文档状态改为 "RUNNING" 之前也读取了同一个文档，这样两个线程会运行相同的处理过程。虽然可以在更新查询中进行状态检查来避免这一问题，但是十分复杂：

```
var cursor = db.processes.find({"status" : "READY"});
cursor.sort({"priority" : -1}).limit(1);
while ((ps = cursor.next()) != null) {
    ps.update({"_id" : ps._id, "status" : "READY"},
              {"$set" : {"status" : "RUNNING"}});
    var lastOp = db.runCommand({getlasterror : 1});
    if (lastOp.n == 1) {
        do_something(ps);
        db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
        break;
    }
```

```
    cursor = db.processes.find({"status" : "READY"});
    cursor.sort({"priority" : -1}).limit(1);
}
```

这样也有问题。因为有先有后，很可能一个线程处理了所有任务，而另外一个就傻傻地呆在那里。A 线程可能会一直占用着进程，B 线程试着抢占失败后，就让 A 线程自己处理所有任务了。

遇到类似这样的情况时，findAndModify 就可大显身手了。findAndModify 能够在一个操作中返回匹配结果并且进行更新。在本例中，处理过程如下所示：

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "update" : {"$set" : {"status" : "RUNNING"}})
{
    "ok" : 1,
    "value" : {
        "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
        "priority" : 1,
        "status" : "READY"
    }
}
```

注意，返回文档中的状态仍然为 "READY"，因为 findAndModify 返回的是修改之前的文档。要是再在集合上进行一次查询，会发现这个文档的 "status" 已经更新成了 "RUNNING"：

```
> db.processes.findOne({"_id" : ps.value._id})
{
    "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
    "priority" : 1,
    "status" : "RUNNING"
}
```

这样的话，程序就变成了下面这样：

```
ps = db.runCommand({"findAndModify" : "processes",
    "query" : {"status" : "READY"},
    "sort" : {"priority" : -1},
    "update" : {"$set" : {"status" : "RUNNING"}}}).value
do_something(ps)
db.process.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
```

findAndModify 可以使用 "update" 键也可以使用 "remove" 键。"remove" 键表示将匹配的文档从集合里面删除。例如，现在不用更新状态了，而是直接删掉，就可以像下面这样：

```
ps = db.runCommand({"findAndModify" : "processes",
    "query" : {"status" : "READY"},
    "sort" : {"priority" : -1},
    "remove" : true}).value
do_something(ps)
```

findAndModify 命令有很多可以使用的字段。

- findAndModify
  字符串，集合名。
- query
  查询文档，用于检索文档的条件。
- sort
  排序结果的条件。
- update
  修改器文档，用于对匹配的文档进行更新（update 和 remove 必须指定一个）。
- remove
  布尔类型，表示是否删除文档（remove 和 update 必须指定一个）。
- new
  布尔类型，表示返回更新前的文档还是更新后的文档。默认是更新前的文档。
- fields
  文档中需要返回的字段（可选）。
- upsert
  布尔类型，值为 true 时表示这是一个 upsert。默认为 false。

"update" 和 "remove" 必须有一个，也只能有一个。要是没有匹配的文档，这个命令会返回一个错误。

3.4 写入安全机制

写入安全（Write Concern）是一种客户端设置，用于控制写入的安全级别。默认情况下，插入、删除和更新都会一直等待数据库响应（写入是否成功），然后才会继续执行。通常，遇到错误时，客户端会抛出一个异常（有些语言中可能不叫“异常”，不过实质上都是类似的东西）。

有一些选项可以用于精确控制需要应用程序等待的内容。两种最基本的写入安全机制是应答式写入（acknowledged wirte）和非应答式写入（unacknowledged write）。应答式写入是默认的方式：数据库会给出响应，告诉你写入操作是否成功执行。非应答式写入不返回任何响应，所以无法知道写入是否成功。

通常来说，应用程序应该使用应答式写入。但是，对于一些不是特别重要的数据（比如日志或者是批量加载数据），你可能不愿意为了自己不关心的数据而等待数据库响应。在这种情况下，可以使用非应答式写入。

尽管非应答式写入不返回数据库错误，但是这不代表应用程序不需要做错误检查。如果尝试向已经关闭的套接字（socket）执行写入，或者写入套接字时发生了错误，都会引起异常。

使用非应答式写入时，一种经常被忽视的错误是插入无效数据。比如，如果试图插入两个具有相同 `"_id"` 字段的文档，shell 就会抛出异常：

```
> db.foo.insert({"_id" : 1})
> db.foo.insert({"_id" : 1})
E11000 duplicate key error index: test.foo.$_id_ dup key: { : 1.0 }
```

如果第二次插入时使用的是非应答式写入，那么第二次插入就不会抛出异常。键重复异常是一种非常常见的错误，还有其他很多类似的错误，比如无效的修改器或者是磁盘空间不足等。

shell 与客户端程序对非应答式写入的实际支持并不一样：shell 在执行非应答式写入后，会检查最后一个操作是否成功，然后才会向用户输出提示信息。因此，如果在集合上执行了一系列无效操作，最后又执行了一个有效操作，shell 并不会提示有错误发生。

```
> db.foo.insert({"_id" : 1}); db.foo.insert({"_id" : 1}); db.foo.count()
1
```

可以调用 `getLastError` 手动强制在 shell 中进行检查，这一操作会检查最后一次操作中的错误。

```
> db.foo.insert({"_id" : 1}); db.foo.insert({"_id" : 1}); print(
... db.getLastError()); db.foo.count()
E11000 duplicate key error index: test.foo.$_id_ dup key: { : 1.0 }
1
```

编写需要在 shell 中执行的脚本时，这是非常有用的。

事实上，还有其他一些写入安全机制，第 11 章会讲述多台服务器之间的写入安全，第 19 章会讲述写入提交。



2012 年，默认的写入安全机制改变了，所以，遗留代码的行为可能会与预期不一致。在此之前，默认的写入是非应答式的。

幸好，很容易得知当前代码是在默认的写入安全机制发生变化之前写的还是之后写的：默认的写入机制变为安全写入之后，所有驱动程序都开始使用 `MongoClient` 这个类。如果程序使用的连接对象是 `Mongo` 或者 `Connection` 或者其他内容，那么这段程序使用的就是旧的、默认不安全的 API。在默认写入安全机制发生变化之前，任何语言都没有使用 `MongoClient` 作为类名，所以，如果你的代码使用了这个类名，说明你的代码是写入安全的。

如果使用的连接不是 `MongoClient`，应在必要时将旧代码中的非应答式写入改成应答式写入。

第 4 章

查询

本章将详细介绍查询。主要会涵盖以下几个方面：

- 使用 `find` 或者 `findOne` 函数和查询文档对数据库执行查询；
- 使用 `$` 条件查询实现范围查询、数据集包含查询、不等式查询，以及其他一些查询；
- 查询将会返回一个数据库游标，游标只会在你需要时才将需要的文档批量返回；
- 还有很多针对游标执行的元操作，包括忽略一定数量的结果，或者限定返回结果的数量，以及对结果排序。

4.1 find简介

MongoDB 中使用 `find` 来进行查询。查询就是返回一个集合中文档的子集，子集的范围从 0 个文档到整个集合。`find` 的第一个参数决定了要返回哪些文档，这个参数是一个文档，用于指定查询条件。

空的查询文档（例如 `{}`）会匹配集合的全部内容。要是不指定查询文档，默认就是 `{}`。例如：

```
> db.c.find()
```

将批量返回集合 c 中的所有文档。

开始向查询文档中添加键 / 值对时，就意味着限定了查询条件。对于绝大多数类型来说，这种方式很简单明了。数值匹配数值，布尔类型匹配布尔类型，字符串匹配

字符串。查询简单的类型，只要指定想要查找的值就好了，十分简单。例如，想要查找 "age" 值为 27 的所有文档，直接将这样的键 / 值对写进查询文档就好了：

```
> db.users.find({"age" : 27})
```

要是想匹配一个字符串，比如值为 "joe" 的 "username" 键，那么直接将键 / 值对写在查询文档中即可：

```
> db.users.find({"username" : "joe"})
```

可以向查询文档加入多个键 / 值对，将多个查询条件组合在一起，这样的查询条件会被解释成“条件 1 AND 条件 2 AND ... AND 条件 N”。例如，要想查询所有用户名为 joe 且年龄为 27 岁的用户，可以像下面这样：

```
> db.users.find({"username" : "joe", "age" : 27})
```

4.1.1　指定需要返回的键

有时并不需要将文档中所有键 / 值对都返回。遇到这种情况，可以通过 find（或者 findOne）的第二个参数来指定想要的键。这样做既会节省传输的数据量，又能节省客户端解码文档的时间和内存消耗。

例如，如果只对用户集合的 "username" 和 "email" 键感兴趣，可以使用如下查询返回这些键：

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
    "_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
    "username" : "joe",
    "email" : "joe@example.com"
}
```

可以看到，默认情况下 "_id" 这个键总是被返回，即便是没有指定要返回这个键。

也可以用第二个参数来剔除查询结果中的某些键 / 值对。例如，文档中有很多键，但是我们不希望结果中含有 "fatal_weakness" 键：

```
> db.users.find({}, {"id" : 0})
```

使用这种方式，也可以把 "_id" 键剔除掉：

```
> db.users.find({}, {"username" : 1, "_id" : 0})
{
    "username" : "joe",
}
```

4.1.2 限制

查询使用上有些限制。传递给数据库的查询文档的值必须是常量。（在你自己的代码里可以是正常的变量。）也就是不能引用文档中其他键的值。例如，要想保持库存，有 `"in_stock"`（剩余库存）和 `"num_sold"`（已出售）两个键，想通过下列查询来比较两者的值是行不通的：

```
> db.stock.find({"in_stock" : "this.num_sold"}) // 这样是行不通的
```

的确有办法实现类似的操作（详见 4.4 节），但通常需要略微修改一下文档结构，就能通过普通查询来完成这样的操作了，这种方式性能更好。在这个例子中，可以在文档中使用 `"initial_stock"`（初始库存）和 `"in_stock"` 两个键。这样，每当有人购买物品，就将 `"in_stock"` 减去 1。这样，只需要用一个简单的查询就能知道哪种商品已脱销：

```
> db.stock.find({"in_stock" : 0})
```

4.2 查询条件

查询不仅能像前面说的那样精确匹配，还能匹配更加复杂的条件，比如范围、OR 子句和取反。

4.2.1 查询条件

`"$lt"`、`"$lte"`、`"$gt"` 和 `"$gte"` 就是全部的比较操作符，分别对应 <、<=、> 和 >=。可以将其组合起来以便查找一个范围的值。例如，查询 18~30 岁（含）的用户，就可以像下面这样：

```
> db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
```

这样就可以查找到 `"age"` 字段大于等于 18、小于等于 30 的所有文档。

这样的范围查询对日期尤为有用。例如，要查找在 2007 年 1 月 1 日前注册的人，可以像下面这样：

```
> start = new Date("01/01/2007")
> db.users.find({"registered" : {"$lt" : start}})
```

可以对日期进行精确匹配，但是用处不大，因为文档中的日期是精确到毫秒的。而我们通常是想得到一天、一周或者是一个月的数据，这样的话，使用范围查询就很有必要了。

对于文档的键值不等于某个特定值的情况，就要使用另外一种条件操作符 "$ne" 了，它表示“不相等”。若是想要查询所有名字不为 joe 的用户，可以像下面这样查询：

```
> db.users.find({"username" : {"$ne" : "joe"}})
```

"$ne" 能用于所有类型的数据。

4.2.2 OR查询

MongoDB 中有两种方式进行 OR 查询："$in" 可以用来查询一个键的多个值；"$or" 更通用一些，可以在多个键中查询任意的给定值。

如果一个键需要与多个值进行匹配的话，就要用 "$in" 操作符，再加一个条件数组。例如，抽奖活动的中奖号码是 725、542 和 390。要找出全部的中奖文档的话，可以构建如下查询：

```
> db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
```

"$in" 非常灵活，可以指定不同类型的条件和值。例如，在逐步将用户的 ID 号迁移成用户名的过程中，查询时需要同时匹配 ID 和用户名：

```
> db.users.find({"user_id" : {"$in" : [12345, "joe"]})
```

这会匹配 "user_id" 等于 12345 的文档，也会匹配 "user_id" 等于 "joe" 的文档。

要是 "$in" 对应的数组只有一个值，那么和直接匹配这个值效果一样。例如，{ticket_no : {$in:[725]}} 和 {ticket_no : 725} 的效果一样。

与 "$in" 相对的是 "$nin"，"$nin" 将返回与数组中所有条件都不匹配的文档。要是想返回所有没有中奖的人，就可以用如下方法进行查询：

```
> db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]}})
```

该查询会返回所有没有中奖的人。

"$in" 能对单个键做 OR 查询，但要是想找到 "ticket_no" 为 725 或者 "winner" 为 true 的文档该怎么办呢？对于这种情况，应该使用 "$or"。"$or" 接受一个包含所有可能条件的数组作为参数。上面中奖的例子如果用 "$or" 改写将是下面这个样子：

```
> db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
```

"$or" 可以包含其他条件。例如，如果希望匹配到中奖的 "ticket_no"，或者 "winner" 键的值为 true 的文档，就可以这么做：

```
> db.raffle.find({"$or" : [{"ticket_no" : {"$in" : [725, 542, 390]}},
                           {"winner" : true}]})
```

使用普通的 AND 型查询时，总是希望尽可能用最少的条件来限定结果的范围。OR 型查询正相反：第一个条件应该尽可能匹配更多的文档，这样才是最为高效的。

`"$or"` 在任何情况下都会正常工作。如果查询优化器可以更高效地处理 `"$in"`，那就选择使用它。

4.2.3 $not

`"$not"` 是元条件句，即可以用在任何其他条件之上。就拿取模运算符 `"$mod"` 来说。`"$mod"` 会将查询的值除以第一个给定值，若余数等于第二个给定值则匹配成功：

```
> db.users.find({"id_num" : {"$mod" : [5, 1]}})
```

上面的查询会返回 `"id_num"` 值为 1、6、11、16 等的用户。但要是想返回 `"id_num"` 为 2、3、4、5、7、8、9、10、12 等的用户，就要用 `"$not"` 了：

```
> db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})
```

`"$not"` 与正则表达式联合使用时极为有用，用来查找那些与特定模式不匹配的文档（4.3.2 节会详细讲述正则表达式的使用）。

4.2.4 条件语义

如果比较一下上一章的更新修改器和前面的查询文档，会发现以 $ 开头的键位于在不同的位置。在查询中，`"$lt"` 在内层文档，而更新中 `"$inc"` 则是外层文档的键。基本可以肯定：条件语句是内层文档的键，而修改器则是外层文档的键。

可以对一个键应用多个条件。例如，要查找年龄为 20~30 的所有用户，可以在 `"age"` 键上使用 `"$gt"` 和 `"$lt"`：

```
> db.users.find({"age" : {"$lt" : 30, "$gt" : 20}})
```

一个键可以有任意多个条件，但是一个键不能对应多个更新修改器。例如，修改器文档不能同时含有 `{"$inc" : {"age" : 1}, "$set" : {age : 40}}`，因为修改了 `"age"` 两次。但是对于查询条件句就没有这种限定。

有一些“元操作符”（meta-operator）也位于外层文档中，比如 `"$and"`、`"$or"` 和 `"$nor"`。它们的使用形式类似：

```
> db.users.find({"$and" : [{"x" : {"$lt" : 1}}, {"x" : 4}]})
```

这个查询会匹配那些 "x" 字段的值小于 1 并且等于 4 的文档。虽然这两个条件看起来是矛盾的，但是这是完全有可能的，比如，如果 "x" 字段的值是这样一个数组 {"x" : [0, 4]}，那么这个文档就与查询条件相匹配。注意，查询优化器不会对 "$and" 进行优化，这与其他操作符不同。如果把上面的查询改成下面这样，效率会更高：

```
> db.users.find({"x" : {"$lt" : 1, "$in" : [4]}})
```

4.3 特定类型的查询

如第 2 章所述，MongoDB 的文档可以使用多种类型的数据。其中有一些在查询时会有特别的表现。

4.3.1 null

null 类型的行为有点奇怪。它确实能匹配自身，所以要是有一个包含如下文档的集合：

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

就可以按照预期的方式查询 "y" 键为 null 的文档：

```
> db.c.find({"y" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
```

但是，null 不仅会匹配某个键的值为 null 的文档，而且还会匹配不包含这个键的文档。所以，这种匹配还会返回缺少这个键的所有文档：

```
> db.c.find({"z" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

如果仅想匹配键值为 null 的文档，既要检查该键的值是否为 null，还要通过 "$exists" 条件判定键值已存在：

```
> db.c.find({"z" : {"$in" : [null], "$exists" : true}})
```

很遗憾，没有 "$eq" 操作符，所以这条查询语句看上去有些令人费解，但是使用只有一个元素的 "$in" 操作符效果是一样的。

4.3.2 正则表达式

正则表达式能够灵活有效地匹配字符串。例如，想要查找所有名为 Joe 或者 joe 的用户，就可以使用正则表达式执行不区分大小写的匹配：

```
> db.users.find({"name" : /joe/i})
```

系统可以接受正则表达式标志（`i`），但不是一定要有。现在已经匹配了各种大小写组合形式的 joe，如果还希望匹配如 `"joey"` 这样的键，可以略微修改一下刚刚的正则表达式：

```
> db.users.find({"name" : /joey?/i})
```

MongoDB 使用 Perl 兼容的正则表达式（PCRE）库来匹配正则表达式，任何 PCRE 支持的正则表达式语法都能被 MongoDB 接受。建议在查询中使用正则表达式前，先在 JavaScript shell 中检查一下语法，确保匹配与设想的一致。



MongoDB 可以为前缀型正则表达式（比如 `/^joey/`）查询创建索引，所以这种类型的查询会非常高效。

正则表达式也可以匹配自身。虽然几乎没有人直接将正则表达式插入到数据库中，但要是万一你这么做了，也可以匹配到自身：

```
> db.foo.insert({"bar" : /baz/})
> db.foo.find({"bar" : /baz/})
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "bar" : /baz/
}
```

4.3.3 查询数组

查询数组元素与查询标量值是一样的。例如，有一个水果列表，如下所示：

```
> db.food.insert({"fruit" : ["apple", "banana", "peach"]})
```

下面的查询：

```
> db.food.find({"fruit" : "banana"})
```

会成功匹配该文档。这个查询好比我们对一个这样的（不合法）文档进行查询：`{"fruit" : "apple", "fruit" : "banana", "fruit" : "peach"}`。

1. $all

如果需要通过多个元素来匹配数组，就要用 "$all" 了。这样就会匹配一组元素。例如，假设创建了一个包含 3 个元素的集合：

```
> db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})
> db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})
> db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})
```

要找到既有 "apple" 又有 "banana" 的文档，可以使用 "$all" 来查询：

```
> db.food.find({fruit : {$all : ["apple", "banana"]}})
      {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
      {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

这里的顺序无关紧要。注意，第二个结果中 "banana" 在 "apple" 之前。要是对只有一个元素的数组使用 "$all"，就和不用 "$all" 一样了。例如，{fruit : {$all : ['apple']} 和 {fruit : 'apple'} 的查询结果完全一样。

也可以使用整个数组进行精确匹配。但是，精确匹配对于缺少元素或者元素冗余的情况就不大灵了。例如，下面的方法会匹配之前的第一个文档：

```
> db.food.find({"fruit" : ["apple", "banana", "peach"]})
```

但是下面这个就不会匹配：

```
> db.food.find({"fruit" : ["apple", "banana"]})
```

这个也不会匹配：

```
> db.food.find({"fruit" : ["banana", "apple", "peach"]})
```

要是想查询数组特定位置的元素，需使用 *key.index* 语法指定下标：

```
> db.food.find({"fruit.2" : "peach"})
```

数组下标都是从 0 开始的，所以上面的表达式会用数组的第 3 个元素和 "peach" 进行匹配。

2. $size

"$size" 对于查询数组来说也是非常有用的，顾名思义，可以用它查询特定长度的数组。例如：

```
> db.food.find({"fruit" : {"$size" : 3}})
```

得到一个长度范围内的文档是一种常见的查询。"$size" 并不能与其他查询条件（比如 "$gt"）组合使用，但是这种查询可以通过在文档中添加一个 "size" 键的方

式来实现。这样每一次向指定数组添加元素时，同时增加 "size" 的值。比如，原本这样的更新：

```
> db.food.update(criteria, {"$push" : {"fruit" : "strawberry"}})
```

就要变成下面这样：

```
> db.food.update(criteria,
... {"$push" : {"fruit" : "strawberry"}, "$inc" : {"size" : 1}})
```

自增操作的速度非常快，所以对性能的影响微乎其微。这样存储文档后，就可以像下面这样查询了：

```
> db.food.find({"size" : {"$gt" : 3}})
```

很遗憾，这种技巧并不能与 "$addToSet" 操作符同时使用。

3. $slice操作符

本章前面已经提及，find 的第二个参数是可选的，可以指定需要返回的键。这个特别的 "$slice" 操作符可以返回某个键匹配的数组元素的一个子集。

例如，假设现在有一个博客文章的文档，我们希望返回前 10 条评论，可以这样做：

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
```

也可以返回后 10 条评论，只要在查询条件中使用 -10 就可以了：

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})
```

"$slice" 也可以指定偏移值以及希望返回的元素数量，来返回元素集合中间位置的某些结果：

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})
```

这个操作会跳过前 23 个元素，返回第 24～33 个元素。如果数组不够 33 个元素，则返回第 23 个元素后面的所有元素。

除非特别声明，否则使用 "$slice" 时将返回文档中的所有键。别的键说明符都是默认不返回未提及的键，这点与 "$slice" 不太一样。例如，有如下博客文章文档：

```
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
```

```
                "name" : "joe",
                "email" : "joe@example.com",
                "content" : "nice post."
            },
            {
                "name" : "bob",
                "email" : "bob@example.com",
                "content" : "good post."
            }
        ]
    }
```

用 "$slice" 来获取最后一条评论，可以这样：

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -1}})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

"title" 和 "content" 都返回了，即便是并没有显式地出现在键说明符中。

4. 返回一个匹配的数组元素

如果知道元素的下标，那么 "$slice" 非常有用。但有时我们希望返回与查询条件相匹配的任意一个数组元素。可以使用 $ 操作符得到一个匹配的元素。对于上面的博客文章示例，可以用如下的方式得到 Bob 的评论：

```
> db.blog.posts.find({"comments.name" : "bob"}, {"comments.$" : 1})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "comments" : [
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

注意，这样只会返回第一个匹配的文档。如果 Bob 在这篇博客文章下写过多条评论，只有 "comments" 数组中的第一条评论会被返回。

5. 数组和范围查询的相互作用

文档中的标量（非数组元素）必须与查询条件中的每一条语句相匹配。例如，如果使用 `{"x" : {"$gt" : 10, "$lt" : 20}}` 进行查询，只会匹配 `"x"` 键的值大于等于 10 并且小于等于 20 的文档。但是，假如某个文档的 `"x"` 字段是一个数组，如果 `"x"` 键的某一个元素与查询条件的任意一条语句相匹配（查询条件中的每条语句可以匹配不同的数组元素），那么这个文档也会被返回。

下面用一个例子来详细说明这种情况。假如有如下所示的文档：

```
{"x" : 5}
{"x" : 15}
{"x" : 25}
{"x" : [5, 25]}
```

如果希望找到 `"x"` 键的值位于 10 和 20 之间的所有文档，直接想到的查询方式是使用 `db.test.find({"x" : {"$gt" : 10, "$lt" : 20}})`，希望这个查询的返回文档是 `{"x" : 15}`。但是，实际返回了两个文档：

```
> db.test.find({"x" : {"$gt" : 10, "$lt" : 20}})
{"x" : 15}
{"x" : [5, 25]}
```

5 和 25 都不位于 10 和 20 之间，但是这个文档也返回了，因为 25 与查询条件中的第一个语句（大于 10）相匹配，5 与查询条件中的第二个语句（小于 20）相匹配。

这使对数组使用范围查询没有用：范围会匹配任意多元素数组。有几种方式可以得到预期的行为。

首先，可以使用 `"$elemMatch"` 要求 MongoDB 同时使用查询条件中的两个语句与一个数组元素进行比较。但是，这里有一个问题，`"$elemMatch"` 不会匹配非数组元素：

```
> db.test.find({"x" : {"$elemMatch" : {"$gt" : 10, "$lt" : 20}})
> // 查不到任何结果
```

`{"x" : 15}` 这个文档与查询条件不再匹配了，因为它的 `"x"` 字段是个数组。

如果当前查询的字段上创建过索引（第 5 章会讲述索引相关内容），可以使用 `min()` 和 `max()` 将查询条件遍历的索引范围限制为 `"$gt"` 和 `"$lt"` 的值：

```
> db.test.find({"x" : {"$gt" : 10, "$lt" : 20}).min({"x" : 10}).max({"x" : 20})
{"x" : 15}
```

现在，这个查询只会遍历值位于 10 和 20 之间的索引，不再与 5 和 25 进行比较。只有当前查询的字段上建立过索引时，才可以使用 `min()` 和 `max()`，而且，必须为这个索引的所有字段指定 `min()` 和 `max()`。

在可能包含数组的文档上应用范围查询时，使用 `min()` 和 `max()` 是非常好的：如果在整个索引范围内对数组使用 `"$gt"`/`"$lt"` 查询，效率是非常低的。查询条件会与所有值进行比较，会查询每一个索引，而不仅仅是指定索引范围内的值。

4.3.4 查询内嵌文档

有两种方法可以查询内嵌文档：查询整个文档，或者只针对其键 / 值对进行查询。

查询整个内嵌文档与普通查询完全相同。例如，有如下文档：

```
{
    "name" : {
        "first" : "Joe",
        "last" : "Schmoe"
    },
    "age" : 45
}
```

要查寻姓名为 Joe Schmoe 的人可以这样：

```
> db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})
```

但是，如果要查询一个完整的子文档，那么子文档必须精确匹配。如果 Joe 决定添加一个代表中间名的键，这个查询就不再可行了，因为查询条件不再与整个内嵌文档相匹配。而且这种查询还是与顺序相关的，`{"last" : "Schmoe", "first" : "Joe"}` 什么都匹配不到。

如果允许的话，通常只针对内嵌文档的特定键值进行查询，这是比较好的做法。这样，即便数据模式改变，也不会导致所有查询因为要精确匹配而一下子都挂掉。我们可以使用点表示法查询内嵌文档的键：

```
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
```

现在，如果 Joe 增加了更多的键，这个查询依然会匹配他的姓和名。

这种点表示法是查询文档区别于其他文档的主要特点。查询文档可以包含点来表达“进入内嵌文档内部”的意思。点表示法也是待插入的文档不能包含“.”的原因。将 URL 作为键保存时经常会遇到此类问题。一种解决方法就是在插入前或者提取后执行一个全局替换，将“.”替换成一个 URL 中的非法字符。

当文档结构变得更加复杂以后，内嵌文档的匹配需要些许技巧。例如，假设有博客文章若干，要找到由 Joe 发表的 5 分以上的评论。博客文章的结构如下例所示：

```
> db.blog.find()
{
    "content" : "...",
    "comments" : [
        {
            "author" : "joe",
            "score" : 3,
            "comment" : "nice post"
        },
        {
            "author" : "mary",
            "score" : 6,
            "comment" : "terrible post"
        }
    ]
}
```

不能直接用 `db.blog.find({"comments" : {"author" : "joe","score" : {"$gte" : 5}}})` 来查寻。内嵌文档的匹配，必须要整个文档完全匹配，而这个查询不会匹配 `"comment"` 键。使用 `db.blog.find({"comments.author" : "joe","comments.score" : {"$gte" : 5}}` 也不行，因为符合 `author` 条件的评论和符合 `score` 条件的评论可能不是同一条评论。也就是说，会返回刚才显示的那个文档。因为 `"author" : "joe"` 在第一条评论中匹配了，`"score" : 6` 在第二条评论中匹配了。

要正确地指定一组条件，而不必指定每个键，就需要使用 `"$elemMatch"`。这种模糊的命名条件句能用来在查询条件中部分指定匹配数组中的单个内嵌文档。所以正确的写法应该是下面这样的：

```
> db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe",
                                              "score" : {"$gte" : 5}}}})
```

`"$elemMatch"` 将限定条件进行分组，仅当需要对一个内嵌文档的多个键操作时才会用到。

4.4 $where查询

键 / 值对是一种表达能力非常好的查询方式，但是依然有些需求它无法表达。其他方法都败下阵时，就轮到 `"$where"` 子句登场了，用它可以在查询中执行任意的 JavaScript。这样就能在查询中做（几乎）任何事情。为安全起见，应该严格限制或者消除 `"$where"` 语句的使用。应该禁止终端用户使用任意的 `"$where"` 语句。

"$where" 语句最常见的应用就是比较文档中的两个键的值是否相等。假如我们有如下文档：

```
> db.foo.insert({"apple" : 1, "banana" : 6, "peach" : 3})
> db.foo.insert({"apple" : 8, "spinach" : 4, "watermelon" : 4})
```

我们希望返回两个键具有相同值的文档。第二个文档中，"spinach" 和 "watermelon" 的值相同，所以需要返回该文档。MongoDB 似乎从来没有提供过一个 $ 条件语句来做这种查询，所以只能用 "$where" 子句借助 JavaScript 来完成了：

```
> db.foo.find({"$where" : function () {
... for (var current in this) {
...     for (var other in this) {
...         if (current != other && this[current] == this[other]) {
...             return true;
...         }
...     }
... }
... return false;
... }});
```

如果函数返回 true，文档就做为结果集的一部分返回；如果为 false，就不返回。

不是非常必要时，一定要避免使用 "$where" 查询，因为它们在速度上要比常规查询慢很多。每个文档都要从 BSON 转换成 JavaScript 对象，然后通过 "$where" 表达式来运行。而且 "$where" 语句不能使用索引，所以只在走投无路时才考虑 "$where" 这种用法。先使用常规查询进行过滤，然后再使用 "$where" 语句，这样组合使用可以降低性能损失。如果可能的话，使用 "$where" 语句前应该先使用索引进行过滤，"$where" 只用于对结果进行进一步过滤。

进行复杂查询的另一种方法是使用聚合工具，第 7 章会详细介绍。

服务器端脚本

在服务器上执行 JavaScript 时必须注意安全性。如果使用不当，服务器端 JavaScript 很容易受到注入攻击，与关系型数据库中的注入攻击类似。不过，只要在接受输入时遵循一些规则，就可以安全地使用 JavaScript。也可以在运行 mongod 时指定 --noscripting 选项，完全关闭 JavaScript 的执行。

JavaScript 的安全问题都与用户在服务器上提供的程序相关。如果希望避免这些风险，那么就要确保不能直接将用户输入的内容传递给 mongod。例如，假如你希望打印一句“Hello, *name*!”，这里的 name 是由用户提供的。使用如下所示的 JavaScript 函数是非常容易想到的：

```
> func = "function() { print('Hello, "+name+"!'); }"
```

如果这里的 name 是一个用户定义的变量，它可能会是 "'); db.dropDatabase(); print('" 这样一个字符串，因此，上面的代码会被转换成如下代码：

```
> func = "function() { print('Hello, '); db.dropDatabase(); print('!'); }"
```

如果执行这段代码，你的整个数据库就会被删除！

为了避免这种情况，应该使用*作用域*来传递 name 的值。以 Python 为例：

```
func = pymongo.code.Code("function() { print('Hello, '+username+'!'); }",
            {"username": name})
```

现在，数据库会输出如下的内容，不会有任何风险：

```
Hello, '); db.dropDatabase(); print('!
```

由于代码实际上可能是字符串和*作用域*的混合体，所以大多数驱动程序都有一种特殊类型，用于向数据库传递代码。作用域是用于表示变量名和值的映射的文档。对于要被执行的 JavaScript 函数来说，这个映射就是一个局部作用域。因此，在上面的例子中，函数可以访问 username 这个变量，这个变量的值就是用户传进来的字符串。



shell 中没有包含作用域的代码类型，所以作用域只能在字符串或者 JavaScript 函数中使用。

4.5 游标

数据库使用游标返回 find 的执行结果。客户端对游标的实现通常能够对最终结果进行有效的控制。可以限制结果的数量，略过部分结果，根据任意键按任意顺序的组合对结果进行各种排序，或者是执行其他一些强大的操作。

要想从 shell 中创建一个游标，首先要对集合填充一些文档，然后对其执行查询，并将结果分配给一个局部变量（用 var 声明的变量就是局部变量）。这里，先创建一个简单的集合，而后做个查询，并用 cursor 变量保存结果：

```
> for(i=0; i<100; i++) {
...     db.collection.insert({x : i});
... }
> var cursor = db.collection.find();
```

这么做的好处是可以一次查看一条结果。如果将结果放在全局变量或者就没有放在变量中，MongoDB shell 会自动迭代，自动显示最开始的若干文档。也就是在这之前我们看到的种种例子，一般大家只想通过 shell 看看集合里面有什么，而不是想在其中实际运行程序，这样设计也就很合适。

要迭代结果，可以使用游标的 `next` 方法。也可以使用 `hasNext` 来查看游标中是否还有其他结果。典型的结果遍历如下所示：

```
> while (cursor.hasNext()) {
...     obj = cursor.next();
...     // do stuff
... }
```

`cursor.hasNext()` 检查是否有后续结果存在，然后用 `cursor.next()` 获得它。

游标类还实现了 JavaScript 的迭代器接口，所以可以在 `forEach` 循环中使用：

```
> var cursor = db.people.find();
> cursor.forEach(function(x) {
...     print(x.name);
... });
adam
matt
zak
```

调用 `find` 时，shell 并不立即查询数据库，而是等待真正开始要求获得结果时才发送查询，这样在执行之前可以给查询附加额外的选项。几乎游标对象的每个方法都返回游标本身，这样就可以按任意顺序组成方法链。例如，下面几种表达是等价的：

```
> var cursor = db.foo.find().sort({"x" : 1}).limit(1).skip(10);
> var cursor = db.foo.find().limit(1).sort({"x" : 1}).skip(10);
> var cursor = db.foo.find().skip(10).limit(1).sort({"x" : 1});
```

此时，查询还没有真正执行，所有这些函数都只是构造查询。现在，假设我们执行如下操作：

```
> cursor.hasNext()
```

这时，查询被发往服务器。shell 立刻获取前 100 个结果或者前 4 MB 数据（两者之中较小者），这样下次调用 `next` 或者 `hasNext` 时就不必再次连接服务器取结果了。客户端用光了第一组结果，shell 会再一次联系数据库，使用 `getMore` 请求提取更多的结果。`getMore` 请求包含一个查询标识符，向数据库询问是否还有更多的结果，如果有，则返回下一批结果。这个过程会一直持续到游标耗尽或者结果全部返回。

4.5.1 limit、skip和sort

最常用的查询选项就是限制返回结果的数量、忽略一定数量的结果以及排序。所有这些选项一定要在查询被发送到服务器之前指定。

要限制结果数量，可在 find 后使用 limit 函数。例如，只返回 3 个结果，可以这样：

```
> db.c.find().limit(3)
```

要是匹配的结果不到 3 个，则返回匹配数量的结果。limit 指定的是上限，而非下限。

skip 与 limit 类似：

```
> db.c.find().skip(3)
```

上面的操作会略过前三个匹配的文档，然后返回余下的文档。如果集合里面能匹配的文档少于 3 个，则不会返回任何文档。

sort 接受一个对象作为参数，这个对象是一组键 / 值对，键对应文档的键名，值代表排序的方向。排序方向可以是 1（升序）或者 –1（降序）。如果指定了多个键，则按照这些键被指定的顺序逐个排序。例如，要按照 "username" 升序及 "age" 降序排序，可以这样写：

```
> db.c.find().sort({username : 1, age : -1})
```

这 3 个方法可以组合使用。这对于分页非常有用。例如，你有个在线商店，有人想搜索 mp3。若是想每页返回 50 个结果，而且按照价格从高到低排序，可以这样写：

```
> db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
```

点击“下一页”可以看到更多的结果，通过 skip 也可以非常简单地实现，只需要略过前 50 个结果就好了（已经在第一页显示了）：

```
> db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})
```

然而，略过过多的结果会导致性能问题，下一小节会讲述如何避免略过大量结果。

比较顺序

MongoDB 处理不同类型的数据是有一定顺序的。有时一个键的值可能是多种类型的，例如，整型和布尔型，或者字符串和 null。如果对这种混合类型的键排序，其排序顺序是预先定义好的。优先级从小到大，其顺序如下：

(1) 最小值；

(2) null；

(3) 数字（整型、长整型、双精度）；

(4) 字符串；

(5) 对象 / 文档；

(6) 数组；

(7) 二进制数据；

(8) 对象 ID；

(9) 布尔型；

(10) 日期型；

(11) 时间戳；

(12) 正则表达式；

(13) 最大值 。

4.5.2 避免使用skip略过大量结果

用 skip 略过少量的文档还是不错的。但是要是数量非常多的话，skip 就会变得很慢，因为要先找到需要被略过的数据，然后再抛弃这些数据。大多数数据库都会在索引中保存更多的元数据，用于处理 skip，但是MongoDB目前还不支持，所以要尽量避免略过太多的数据。通常可以利用上次的结果来计算下一次查询条件。

1. 不用skip对结果分页

最简单的分页方法就是用 limit 返回结果的第一页，然后将每个后续页面作为相对于开始的偏移量返回。

```
> // 不要这么用：略过的数据比较多时，速度会变得很慢
> var page1 = db.foo.find(criteria).limit(100)
> var page2 = db.foo.find(criteria).skip(100).limit(100)
> var page3 = db.foo.find(criteria).skip(200).limit(100)
...
```

然而，一般来讲可以找到一种方法在不使用 skip 的情况下实现分页，这取决于查询本身。例如，要按照 "date" 降序显示文档列表。可以用如下方式获取结果的第一页：

```
> var page1 = db.foo.find().sort({"date" : -1}).limit(100)
```

然后，可以利用最后一个文档中 "date" 的值作为查询条件，来获取下一页：

```
var latest = null;
```

```
// 显示第一页
while (page1.hasNext()) {
   latest = page1.next();
   display(latest);
}

// 获取下一页
var page2 = db.foo.find({"date" : {"$gt" : latest.date}});
page2.sort({"date" : -1}).limit(100);
```

这样查询中就没有 skip 了。

2. 随机选取文档

从集合里面随机挑选一个文档算是个常见问题。最笨的（也很慢的）做法就是先计算文档总数，然后选择一个从 0 到文档数量之间的随机数，利用 find 做一次查询，略过这个随机数那么多的文档，这个随机数的取值范围为 0 到集合中文档的总数：

```
> // 不要这么用
> var total = db.foo.count()
> var random = Math.floor(Math.random()*total)
> db.foo.find().skip(random).limit(1)
```

这种选取随机文档的做法效率太低：首先得计算总数（要是有查询条件就会很费时），然后用 skip 略过大量结果也会非常耗时。

略微动动脑筋，从集合里面查找一个随机元素还是有好得多的办法的。秘诀就是在插入文档时给每个文档都添加一个额外的随机键。例如在 shell 中，可以用 Math.random()（产生一个 0~1 的随机数）：

```
> db.people.insert({"name" : "joe", "random" : Math.random()})
> db.people.insert({"name" : "john", "random" : Math.random()})
> db.people.insert({"name" : "jim", "random" : Math.random()})
```

这样，想要从集合中查找一个随机文档，只要计算一个随机数并将其作为查询条件就好了，完全不用 skip：

```
> var random = Math.random()
> result = db.foo.findOne({"random" : {"$gt" : random}})
```

偶尔也会遇到产生的随机数比集合中所有随机值都大的情况，这时就没有结果返回了。遇到这种情况，那就将条件操作符换一个方向：

```
> if (result == null) {
...     result = db.foo.findOne({"random" : {"$lt" : random}})
... }
```

要是集合里面本就没有文档，则会返回 null，这说得通。

这种技巧还可以和其他各种复杂的查询一同使用，仅需要确保有包含随机键的索引

即可。例如，想在加州随机找一个水暖工，可以对 "profession"、"state" 和 "random" 建立索引：

```
> db.people.ensureIndex({"profession" : 1, "state" : 1, "random" : 1})
```

这样就能很快得出一个随机结果（关于索引，详见第 5 章）。

4.5.3 高级查询选项

有两种类型的查询：简单查询（plain query）和封装查询（wrapped query）。简单查询就像下面这样：

```
> var cursor = db.foo.find({"foo" : "bar"})
```

有一些选项可以用于对查询进行"封装"。例如，假设我们执行一个排序：

```
> var cursor = db.foo.find({"foo" : "bar"}).sort({"x" : 1})
```

实际情况不是将 {"foo" : "bar"} 作为查询直接发送给数据库，而是先将查询封装在一个更大的文档中。shell 会把查询从 {"foo" : "bar"} 转换成 {"$query" : {"foo" : "bar"}, "$orderby" : {"x" : 1}}。

绝大多数驱动程序都提供了辅助函数，用于向查询中添加各种选项。下面列举了其他一些有用的选项。

- $maxscan : *integer*
  指定本次查询中扫描文档数量的上限。

  ```
  > db.foo.find(criteria)._addSpecial("$maxscan", 20)
  ```

  如果不希望查询耗时太多，也不确定集合中到底有多少文档需要扫描，那么可以使用这个选项。这样就会将查询结果限定为与被扫描的集合部分相匹配的文档。这种方式的一个坏处是，某些你希望得到的文档没有扫描到。

- $min : *document*
  查询的开始条件。在这样的查询中，文档必须与索引的键完全匹配。查询中会强制使用给定的索引。

  在内部使用时，通常应该使用 "$gt" 代替 "$min"。可以使用 "$min" 强制指定一次索引扫描的下边界，这在复杂查询中非常有用。

- $max : *document*
  查询的结束条件。在这样的查询中，文档必须与索引的键完全匹配。查询中会强制使用给定的索引。

在内部使用时，通常应该使用 "$lg" 而不是 "$max"。可以使用 "$max" 强制指定一次索引扫描的上边界，这在复杂查询中非常有用。

- `$showDiskLoc : true`
  在查询结果中添加一个 "$diskLoc" 字段，用于显示该条结果在磁盘上的位置。例如：

```
> db.foo.find()._addSpecial('$showDiskLoc',true)
{ "_id" : 0, "$diskLoc" : { "file" : 2, "offset" : 154812592 } }
{ "_id" : 1, "$diskLoc" : { "file" : 2, "offset" : 154812628 } }
```

  文件号码显示了这个文档所在的文件。如果这里使用的是 test 数据库，那么这个文档就在 test.2 文件中。第二个字段显示的是该文档在文件中的偏移量。

4.5.4 获取一致结果

数据处理通常的做法就是先把数据从 MongoDB 中取出来，然后做一些变换，最后再存回去：

```
cursor = db.foo.find();

while (cursor.hasNext()) {
    var doc = cursor.next();
    doc = process(doc);
    db.foo.save(doc);
}
```

结果比较少，这样是没问题的，但是如果结果集比较大，MongoDB 可能会多次返回同一个文档。为什么呢？想象一下文档究竟是如何存储的吧。可以将集合看做一个文档列表，如图 4-1 所示。雪花代表文档，因为每一个文档都是美丽且唯一的。

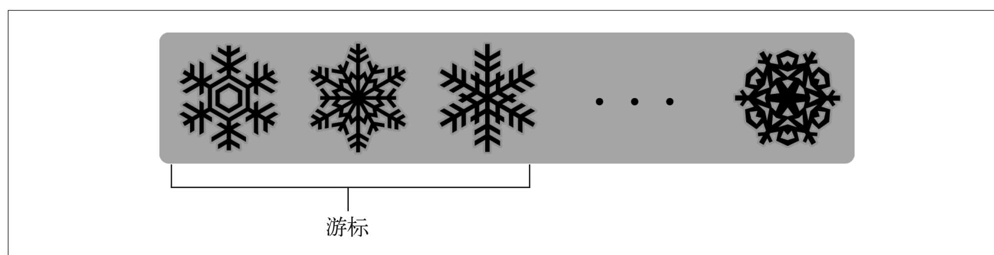


图 4-1：待查询的集合

这样，进行查找时，从集合的开头返回结果，游标不断向右移动。程序获取前 100 个文档并处理。将这些文档保存回数据库时，如果文档体积增加了，而预留空间不足，如图 4-2 所示，这时就需要对体积增大后的文档进行移动。通常会将它们挪至

集合的末尾处（如图 4-3 所示）。

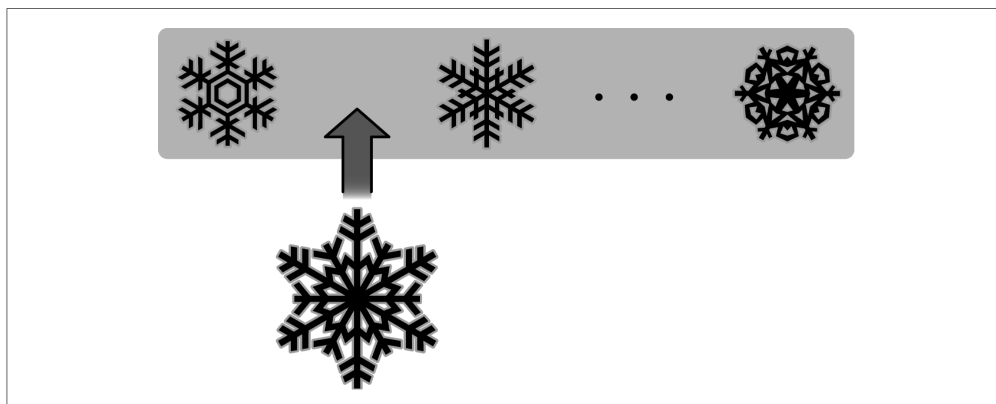


图 4-2：体积变大的文档，可能无法保存回原先的位置

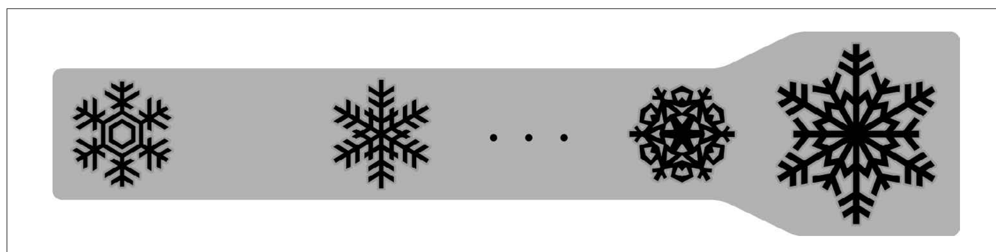


图 4-3：MongoDB 会为更新后无法放回原位置的文档重新分配存储空间

现在，程序继续获取大量的文档，如此往复。当游标移动到集合末尾时，就会返回因体积太大无法放回原位置而被移动到集合末尾的文档，如图 4-4 所示。



图 4-4：游标可能会返回那些由于体积变大而被移动到集合末尾的文档

应对这个问题的方法就是对查询进行快照（snapshot）。如果使用了这个选项，查询就在 "_id" 索引上遍历执行，这样可以保证每个文档只被返回一次。例如，将

`db.foo.find()` 改为：

```
> db.foo.find().snapshot()
```

快照会使查询变慢，所以应该只在必要时使用快照。例如，mongodump（用于备份，第 22 章会介绍）默认在快照上使用查询。

所有返回单批结果的查询都被有效地进行了快照。当游标正在等待获取下一批结果时，如果集合发生了变化，数据才可能出现不一致。

4.5.5 游标生命周期

看待游标有两种角度：客户端的游标以及客户端游标表示的数据库游标。前面讨论的都是客户端的游标，接下来简要看看服务器端发生了什么。

在服务器端，游标消耗内存和其他资源。游标遍历尽了结果以后，或者客户端发来消息要求终止，数据库将会释放这些资源。释放的资源可以被数据库另作他用，这是非常有益的，所以要尽量保证尽快释放游标（在合理的前提下）。

还有一些情况导致游标终止（随后被清理）。首先，游标完成匹配结果的迭代时，它会清除自身。另外，如果客户端的游标已经不在作用域内了，驱动程序会向服务器发送一条特别的消息，让其销毁游标。最后，即便用户没有迭代完所有结果，并且游标也还在作用域中，如果一个游标在 10 分钟内没有使用的话，数据库游标也会自动销毁。这样的话，如果客户端崩溃或者出错，MongoDB 就不需要维护这上千个被打开却不再使用的游标。

这种“超时销毁”的行为是我们希望的：极少有应用程序希望用户花费数分钟坐在那里等待结果。然而，有时的确希望游标持续的时间长一些。若是如此的话，多数驱动程序都实现了一个叫 `immortal` 的函数，或者类似的机制，来告知数据库不要让游标超时。如果关闭了游标的超时时间，则一定要迭代完所有结果，或者主动将其销毁，以确保游标被关闭。否则它会一直在数据库中消耗服务器资源。

4.6 数据库命令

有一种非常特殊的查询类型叫作*数据库命令*（database command）。前面已经介绍过文档的创建、更新、删除以及查询。这些都是数据库命令的使用范畴，包括管理性的任务（比如关闭服务器和克隆数据库）、统计集合内的文档数量以及执行聚合等。

本节主要讲述数据库命令，在数据操作、管理以及监控中，数据库命令都是非常有用的。例如，删除集合是使用 `"drop"` 数据库命令完成的：

```
> db.runCommand({"drop" : "test"});
{
    "nIndexesWas" : 1,
    "msg" : "indexes dropped for collection",
    "ns" : "test.test",
    "ok" : true
}
```

也许你对 shell 辅助函数比较熟悉，这些辅助函数封装数据库命令，并提供更加简单的接口：

```
> db.test.drop()
```

通常，只使用 shell 辅助函数就可以了，但是了解它们底层的命令很有帮助。尤其是当使用旧版本的 shell 连接到新版本的数据库上时，这个 shell 可能不支持新版数据库的一些命令，这时候就不得不直接使用 `runCommand()`。

在前面的章节中已经看到过一些命令了，比如，第 3 章使用 `getLastError` 来查看更新操作影响到的文档数量：

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

本节会更深入地介绍数据库命令，一起来看看这些数据库命令到底是什么，到底是怎么实现的。本节也会介绍 MongoDB 提供的一些非常有用的命令。在 shell 中运行 `db.listCommands()` 可以看到所有的数据库命令。

数据库命令工作原理

数据库命令总会返回一个包含 `"ok"` 键的文档。如果 `"ok"` 的值是 1，说明命令执行成功了；如果值是 0，说明由于一些原因，命令执行失败。

如果 `"ok"` 的值是 0，那么命令的返回文档中就会有一个额外的键 `"errmsg"`。它的值是一个字符串，用于描述命令的失败原因。例如，如果试着在上一节已经删除的集合上再次执行 `drop` 命令：

```
> db.runCommand({"drop" : "test"});
{ "errmsg" : "ns not found", "ok" : false }
```

MongoDB 中的命令被实现为一种特殊类型的查询，这些特殊的查询会在 $cmd 集合

上执行。runCommand 只是接受一个命令文档，并且执行与这个命令文档等价的查询。于是，drop 命令会被转换为如下代码：

```
db.$cmd.findOne({"drop" : "test"});
```

当 MongoDB 服务器得到一个在 $cmd 集合上的查询时，不会对这个查询进行通常的查询处理，而是会使用特殊的逻辑对其进行处理。几乎所有的 MongoDB 驱动程序都会提供一个类似 runCommand 的辅助函数，用于执行命令，而且命令总是能够以简单查询的方式执行。

有些命令需要有管理员权限，而且要在 admin 数据库上才能执行。如果在其他数据库上执行这样的命令，就会得到一个 "access denied"（访问被拒绝）错误。如果当前位于其他的数据库，但是需要执行一个管理员命令，可以使用 adminCommand 而不是 runCommand：

```
> use temp
switched to db temp
> db.runCommand({shutdown:1})
{ "errmsg" : "access denied; use admin db", "ok" : 0 }
> db.adminCommand({"shutdown" : 1})
```

MongoDB 中，数据库命令是少数与字段顺序相关的地方之一：命令名称必须是命令中的第一个字段。因此，{"getLastError" : 1, "w" : 2} 是有效的命令，而 {"w" : 2, "getLastError" : 1} 不是。

第二部分

设计应用

第5章
索引

本章介绍 MongoDB 的索引，索引可以用来优化查询，而且在某些特定类型的查询中，索引是必不可少的。

- 什么是索引？为什么要用索引？
- 如何选择需要建立索引的字段？
- 如何强制使用索引？如何评估索引的效率？
- 创建索引和删除索引。

为集合选择合适的索引是提高性能的关键。

5.1 索引简介

数据库索引与书籍的索引类似。有了索引就不需要翻整本书，数据库可以直接在索引中查找，在索引中找到条目以后，就可以直接跳转到目标文档的位置，这能使查找速度提高几个数量级。

不使用索引的查询称为*全表扫描*（这个术语来自关系型数据库），也就是说，服务器必须查找完一整本书才能找到查询结果。这个处理过程与我们在一本没有索引的书中查找信息很像：从第 1 页开始一直读完整本书。通常来说，应该尽量避免全表扫描，因为对于大集合来说，全表扫描的效率非常低。

来看一个例子，我们创建了一个拥有 1 000 000 个文档的集合（如果你想要 10 000 000 或者 100 000 000 个文档也行，只要你有那个耐心）：

```
> for (i=0; i<1000000; i++) {
...     db.users.insert(
...         {
...             "i" : i,
...             "username" : "user"+i,
...             "age" : Math.floor(Math.random()*120),
...             "created" : new Date()
...         }
...     );
... }
```

如果在这个集合上做查询，可以使用 `explain()` 函数查看 MongoDB 在执行查询的过程中所做的事情。下面试着查询一个随机的用户名：

```
> db.users.find({username: "user101"}).explain()
{
    "cursor" : "BasicCursor",
    "nscanned" : 1000000,
    "nscannedObjects" : 1000000,
    "n" : 1,
    "millis" : 721,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "isMultiKey" : false,
    "indexOnly" : false,
    "indexBounds" : {

    }
}
```

5.2 节会详细介绍输出信息里的这些字段，目前来说可以忽略大多数字段。`"nscanned"` 是 MongoDB 在完成这个查询的过程中扫描的文档总数。可以看到，这个集合中的每个文档都被扫描过了。也就是说，为了完成这个查询，MongoDB 查看了每一个文档中的每一个字段。这个查询耗费了将近 1 秒的时间才完成：`"millis"` 字段显示的是这个查询耗费的毫秒数。

字段 `"n"` 显示了查询结果的数量，这里是 1，因为这个集合中确实只有一个 `username` 为 `"user101"` 的文档。注意，由于不知道集合里的 `username` 字段是唯一的，MongoDB 不得不查看集合中的每一个文档。为了优化查询，将查询结果限制为 1，这样 MongoDB 在找到一个文档之后就会停止了：

```
> db.users.find({username: "user101"}).limit(1).explain()
{
    "cursor" : "BasicCursor",
    "nscanned" : 102,
    "nscannedObjects" : 102,
    "n" : 1,
    "millis" : 2,
    "nYields" : 0,
```

```
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {

        }
    }
```

现在，所扫描的文档数量极大地减少了，而且整个查询几乎是瞬间完成的。但是，这个方案是不现实的：如果要查找的是 `user999999` 呢？我们仍然不得不遍历整个集合，而且，随着用户的增加，查询会越来越慢。

对于此类查询，索引是一个非常好的解决方案：索引可以根据给定的字段组织数据，让 MongoDB 能够非常快地找到目标文档。下面尝试在 `username` 字段上创建一个索引：

```
> db.users.ensureIndex({"username" : 1})
```

由于机器性能和集合大小的不同，创建索引有可能需要花几分钟时间。如果对 `ensureIndex` 的调用没能在几秒钟后返回，可以在另一个 shell 中执行 `db.currentOp()` 或者是检查 mongod 的日志来查看索引创建的进度。

索引创建完成之后，再次执行最初的查询：

```
> db.users.find({"username" : "user101"}).explain()
{
    "cursor" : "BtreeCursor username_1",
    "nscanned" : 1,
    "nscannedObjects" : 1,
    "n" : 1,
    "millis" : 3,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "isMultiKey" : false,
    "indexOnly" : false,
    "indexBounds" : {
        "username" : [
            [
                "user101",
                "user101"
            ]
        ]
    }
}
```

这次 `explain()` 的输出内容比之前复杂一些，但是目前我们只需要注意 `"n"`、`"nscanned"` 和 `"millis"` 这几个字段，可以忽略其他字段。可以看到，这个查询现在几乎是瞬间完成的（甚至可以更好），而且对于任意 `username` 的查询，所耗费

的时间基本一致：

```
> db.users.find({username: "user999999"}).explain().millis
1
```

可以看到，使用了索引的查询几乎可以瞬间完成，这是非常激动人心的。然而，使用索引是有代价的：对于添加的每一个索引，每次写操作（插入、更新、删除）都将耗费更多的时间。这是因为，当数据发生变动时，MongoDB 不仅要更新文档，还要更新集合上的所有索引。因此，MongoDB 限制每个集合上最多只能有 64 个索引。通常，在一个特定的集合上，不应该拥有两个以上的索引。于是，挑选合适的字段建立索引非常重要。



MongoDB 的索引几乎与传统的关系型数据库索引一模一样，所以如果已经掌握了那些技巧，则可以跳过本节的语法说明。后面会介绍一些索引的基础知识，但一定要记住这里涉及的只是冰山一角。绝大多数优化 MySQL/Oracle/SQLite 索引的技巧同样也适用于 MongoDB（包括“Use the Index，Luke”上的教程 http://use-the-index-luke.com）。

为了选择合适的键来建立索引，可以查看常用的查询，以及那些需要被优化的查询，从中找出一组常用的键。例如，在上面的例子中，查询是在 "username" 上进行的。如果这是一个非常通用的查询，或者这个查询造成了性能瓶颈，那么在 "username" 上建立索引会是非常好的选择。然而，如果这只是一个很少用到的查询，或者只是给管理员用的查询（管理员并不需要太在意查询耗费的时间），那就不应该对 "username" 建立索引。

5.1.1 复合索引简介

索引的值是按一定顺序排列的，因此，使用索引键对文档进行排序非常快。然而，只有在首先使用索引键进行排序时，索引才有用。例如，在下面的排序里，"username" 上的索引没什么作用：

```
> db.users.find().sort({"age" : 1, "username" : 1})
```

这里先根据 "age" 排序再根据 "username" 排序，所以 "username" 在这里发挥的作用并不大。为了优化这个排序，可能需要在 "age" 和 "username" 上建立索引：

```
> db.users.ensureIndex({"age" : 1, "username" : 1})
```

这样就建立了一个*复合索引*（compound index）。如果查询中有多个排序方向或者查询条件中有多个键，这个索引就会非常有用。复合索引就是一个建立在多个字段上

的索引。

假如我们有一个 users 集合（如下所示），如果在这个集合上执行一个不排序（称为*自然顺序*）的查询：

```
> db.users.find({}, {"_id" : 0, "i" : 0, "created" : 0})
{ "username" : "user0", "age" : 69 }
{ "username" : "user1", "age" : 50 }
{ "username" : "user2", "age" : 88 }
{ "username" : "user3", "age" : 52 }
{ "username" : "user4", "age" : 74 }
{ "username" : "user5", "age" : 104 }
{ "username" : "user6", "age" : 59 }
{ "username" : "user7", "age" : 102 }
{ "username" : "user8", "age" : 94 }
{ "username" : "user9", "age" : 7 }
{ "username" : "user10", "age" : 80 }
...
```

如果使用 `{"age" : 1，"username" : 1}` 建立索引，这个索引大致会是这个样子：

```
[0, "user100309"] -> 0x0c965148
[0, "user100334"] -> 0xf51f818e
[0, "user100479"] -> 0x00fd7934
...
[0, "user99985" ] -> 0xd246648f
[1, "user100156"] -> 0xf78d5bdd
[1, "user100187"] -> 0x68ab28bd
[1, "user100192"] -> 0x5c7fb621
...
[1, "user999920"] -> 0x67ded4b7
[2, "user100141"] -> 0x3996dd46
[2, "user100149"] -> 0xfce68412
[2, "user100223"] -> 0x91106e23
...
```

每一个索引条目都包含一个 `"age"` 字段和一个 `"username"` 字段，并且指向文档在磁盘上的存储位置（这里使用十六进制数字表示，可以忽略）。注意，这里的 `"age"` 字段是严格升序排列的，`"age"` 相同的条目按照 `"username"` 升序排列。每个 `"age"` 都有大约 8000 个对应的 `"username"`，这里只是挑选了少量数据用于传达大概的信息。

MongoDB 对这个索引的使用方式取决于查询的类型。下面是三种主要的方式。

- `db.users.find({"age" : 21}).sort({"username" : -1})`
  这是一个*点查询*（point query），用于查找单个值（尽管包含这个值的文档可能有多个）。由于索引中的第二个字段，查询结果已经是有序的了：MongoDB 可以从 `{"age" : 21}` 匹配的最后一个索引开始，逆序依次遍历索引：

```
[21, "user999977"] -> 0x9b3160cf
[21, "user999954"] -> 0xfe039231
[21, "user999902"] -> 0x719996aa
...
```

这种类型的查询是非常高效的：MongoDB 能够直接定位到正确的年龄，而且不需要对结果进行排序（因为只需要对数据进行逆序遍历就可以得到正确的顺序了）。

注意，排序方向并不重要：MongoDB 可以在任意方向上对索引进行遍历。

- `db.users.find({"age" : {"$gte" : 21, "$lte" : 30}})`
  这是一个多值查询（multi-value query），查找到多个值相匹配的文档（在本例中，年龄必须介于 21 到 30 之间）。MongoDB 会使用索引中的第一个键 `"age"` 得到匹配的文档，如下所示：

```
[21, "user100000"] -> 0x37555a81
[21, "user100069"] -> 0x6951d16f
[21, "user1001"]   -> 0x9a1f5e0c
[21, "user100253"] -> 0xd54bd959
[21, "user100409"] -> 0x824fef6c
[21, "user100469"] -> 0x5fba778b
...
[30, "user999775"] -> 0x45182d8c
[30, "user999850"] -> 0x1df279e9
[30, "user999936"] -> 0x525caa57
```

通常来说，如果 MongoDB 使用索引进行查询，那么查询结果文档通常是按照索引顺序排列的。

- `db.users.find({"age" : {"$gte" : 21, "$lte" :30}}).sort({"username":1})`
  这是一个多值查询，与上一个类似，只是这次需要对查询结果进行排序。跟之前一样，MongoDB 会使用索引来匹配查询条件：

```
[21, "user100000"] -> 0x37555a81
[21, "user100069"] -> 0x6951d16f
[21, "user1001"]   -> 0x9a1f5e0c
[21, "user100253"] -> 0xd54bd959
...
[22, "user100004"] -> 0x81e862c5
[22, "user100328"] -> 0x83376384
[22, "user100335"] -> 0x55932943
[22, "user100405"] -> 0x20e7e664
...
```

然而，使用这个索引得到的结果集中 `"username"` 是无序的，而查询要求结果以 `"username"` 升序排列，所以 MongoDB 需要先在内存中对结果进行排序，然后才能返回。因此，这个查询通常不如上一个高效。

当然，查询速度取决于有多少个文档与查询条件匹配：如果结果集中只有少数几个文档，MongoDB 对这些文档进行排序并不需要耗费多少时间。如果结果集中的文档数量比较多，查询速度就会比较慢，甚至根本不能用：如果结果集的大小超过 32 MB，MongoDB 就会出错，拒绝对如此多的数据进行排序：

```
Mon Oct 29 16:25:26 uncaught exception: error: {
    "$err" : "too much data for sort() with no index. add an index or
        specify a smaller limit",
    "code" : 10128
}
```

最后一个例子中，还可以使用另一个索引（同样的键，但是顺序调换了）：`{"username" : 1, "age" : 1}`。MongoDB 会反转所有的索引条目，但是会以你期望的顺序返回。MongoDB 会根据索引中的 `"age"` 部分挑选出匹配的文档：

```
["user0", 69]
["user1", 50]
["user10", 80]
["user100", 48]
["user1000", 111]
["user10000", 98]
["user100000", 21] -> 0x73f0b48d
["user100001", 60]
["user100002", 82]
["user100003", 27] -> 0x0078f55f
["user100004", 22] -> 0x5f0d3088
["user100005", 95]
...
```

这样非常好，因为不需要在内存中对大量数据进行排序。但是，MongoDB 不得不扫描整个索引以便找到所有匹配的文档。因此，如果对查询结果的范围做了限制，那么 MongoDB 在几次匹配之后就可以不再扫描索引，在这种情况下，将排序键放在第一位是一个非常好的策略。

可以通过 `explain()` 来查看 MongoDB 对 `db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" : 1})` 的默认行为：

```
 > db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).
... sort({"username" : 1}).
... explain()
{
    "cursor" : "BtreeCursor age_1_username_1",
    "isMultiKey" : false,
    "n" : 83484,
    "nscannedObjects" : 83484,
    "nscanned" : 83484,
    "nscannedObjectsAllPlans" : 83484,
    "nscannedAllPlans" : 83484,
```

```
        "scanAndOrder" : true,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 2766,
        "indexBounds" : {
            "age" : [
                [
                    21,
                    30
                ]
            ],
            "username" : [
                [
                    {
                        "$minElement" : 1
                    },
                    {
                        "$maxElement" : 1
                    }
                ]
            ]
        },
        "server" : "spock:27017"
    }
```

可以忽略大部分字段，后面会有相关介绍。注意，"cursor"字段说明这次查询使用的索引是{"age" : 1, "user name" : 1}，而且只查找了不到1/10的文档（"nscanned"只有83484），但是这个查询耗费了差不多3秒的时间（"millis"字段显示的是毫秒数）。这里的"scanAndOrder"字段的值是true：说明MongoDB必须在内存中对数据进行排序，如之前所述。

可以通过hint来强制MongoDB使用某个特定的索引，再次执行这个查询，但是这次使用{"username" : 1, "age" : 1}作为索引。这个查询扫描的文档比较多，但是不需要在内存中对数据排序：

```
> db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).
... sort({"username" : 1}).
... hint({"username" : 1, "age" : 1}).
... explain()
{
    "cursor" : "BtreeCursor username_1_age_1",
    "isMultiKey" : false,
    "n" : 83484,
    "nscannedObjects" : 83484,
    "nscanned" : 984434,
    "nscannedObjectsAllPlans" : 83484,
    "nscannedAllPlans" : 984434,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 0,
```

```
        "nChunkSkips" : 0,
        "millis" : 14820,
        "indexBounds" : {
            "username" : [
                [
                    {
                        "$minElement" : 1
                    },
                    {
                        "$maxElement" : 1
                    }
                ]
            ],
            "age" : [
                [
                    21,
                    30
                ]
            ]
        },
        "server" : "spock:27017"
    }
```

注意，这次查询耗费了将近 15 秒才完成。对比鲜明，第一个索引速度更快。然而，如果限制每次查询的结果数量，新的赢家产生了：

```
> db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).
... sort({"username" : 1}).
... limit(1000).
... hint({"age" : 1, "username" : 1}).
... explain()['millis']
2031
> db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).
... sort({"username" : 1}).
... limit(1000).
... hint({"username" : 1, "age" : 1}).
... explain()['millis']
181
```

第一个查询耗费的时间仍然介于 2 秒到 3 秒之间，但是第二个查询只用了不到 1/5 秒！因此，应该就在应用程序使用的查询上执行 `explain()`。排除掉那些可能会导致 `explain()` 输出信息不准确的选项。

在实际的应用程序中，`{"`*`sortKey`*`" : 1, "`*`queryCriteria`*`" : 1}` 索引通常是很有用的，因为大多数应用程序在一次查询中只需要得到查询结果最前面的少数结果，而不是所有可能的结果。而且，由于索引在内部的组织形式，这种方式非常易于扩展。索引本质上是树，最小的值在最左边的叶子上，最大的值在最右边的叶子上。如果有一个日期类型的 `"`*`sortKey`*`"`（或是其他能够随时间增加的值），当从左向右遍历这棵树时，你实际上也花费了时间。因此，如果应用程序需要使用最近

数据的机会多于较老的数据，那么 MongoDB 只需在内存中保留这棵树最右侧的分支（最近的数据），而不必将整棵树留在内存中。类似这样的索引是*右平衡的*（right balanced），应该尽可能让索引是右平衡的。"_id" 索引就是一个典型的右平衡索引。

5.1.2　使用复合索引

在多个键上建立的索引就是*复合索引*，在上面的小节中，已经使用过复合索引。复合索引比单键索引要复杂一些，但是也更强大。本节会更深入地介绍复合索引。

1. 选择键的方向

到目前为止，我们的所有索引都是升序的（或者是从最小到最大）。但是，如果需要在两个（或者更多）查询条件上进行排序，可能需要让索引键的方向不同。例如，假设我们要根据年龄从小到大，用户名从 Z 到 A 对上面的集合进行排序。对于这个问题，之前的索引变得不再高效：每一个年龄分组内都是按照 "username" 升序排列的，是 A 到 Z，不是 Z 到 A。对于按 "age" 升序排列按 "username" 降序排列这样的需求来说，用上面的索引得到的数据的顺序没什么用。

为了在不同方向上优化这个复合排序，需要使用与方向相匹配的索引。在这个例子中，可以使用 {"age" : 1，"username" : -1}，它会以下面的方式组织数据：

```
[21, "user999977"] -> 0xe57bf737
[21, "user999954"] -> 0x8bffa512
[21, "user999902"] -> 0x9e1447d1
[21, "user999900"] -> 0x3a6a8426
[21, "user999874"] -> 0xc353ee06
...
[30, "user999936"] -> 0x7f39a81a
[30, "user999850"] -> 0xa979e136
[30, "user999775"] -> 0x5de6b77a
...
[30, "user100324"] -> 0xe14f8e4d
[30, "user100140"] -> 0x0f34d446
[30, "user100050"] -> 0x223c35b1
```

年龄按照从年轻到年长顺序排列，在每一个年龄分组中，用户名是从 Z 到 A 排列的（对于我们的用户名来说，也可以说是按照 "9" 到 "0" 排列的）。

如果应用程序同时需要按照 {"age" : 1，"username" : 1} 优化排序，我们还需要创建一个这个方向上的索引。至于索引使用的方向，与排序方向相同就可以了。注意，相互反转（在每个方向都乘以 –1）的索引是等价的：{"age" : 1， "username" : -1} 适用的查询与 {"age" : -1，"username" : 1} 是完全一样的。

只有基于多个查询条件进行排序时，索引方向才是比较重要的。如果只是基于单一

键进行排序，MongoDB 可以简单地从相反方向读取索引。例如，如果有一个基于 `{"age" : -1}` 的排序和一个基于 `{"age" : 1}` 的索引，MongoDB 会在使用索引时进行优化，就如同存在一个 `{"age" : -1}` 索引一样（所以不要创建两个这样的索引！）。只有在基于多键排序时，方向才变得重要。

2. 使用覆盖索引（covered index）

在上面的例子中，查询只是用来查找正确的文档，然后按照指示获取实际的文档。然后，如果你的查询只需要查找索引中包含的字段，那就根本没必要获取实际的文档。当一个索引包含用户请求的所有字段，可以认为这个索引*覆盖*了本次查询。在实际中，应该优先使用覆盖索引，而不是去获取实际的文档。这样可以保证工作集比较小，尤其与右平衡索引一起使用时。

为了确保查询只使用索引就可以完成，应该使用投射（详见 4.1.1 节）来指定不要返回 `"_id"` 字段（除非它是索引的一部分）。可能还需要对不需要查询的字段做索引，因此需要在编写时就在所需的查询速度和这种方式带来的开销之间做好权衡。

如果在覆盖索引上执行 `explain()`，`"indexOnly"` 字段的值要为 `true`。

如果在一个含有数组的字段上做索引，这个索引永远也无法覆盖查询（因为数组是被保存在索引中的，5.1.4 节会深入介绍）。即便将数组字段从需要返回的字段中剔除，这样的索引仍然无法覆盖查询。

3. 隐式索引

复合索引具有双重功能，而且对不同的查询可以表现为不同的索引。如果有一个 `{"age" : 1, "username" : 1}` 索引，`"age"` 字段会被自动排序，就好像有一个 `{"age" : 1}` 索引一样。因此，这个复合索引可以当作 `{"age" : 1}` 索引一样使用。

这个可以根据需要推广到尽可能多的键：如果有一个拥有 `N` 个键的索引，那么你同时"免费"得到了所有这 `N` 个键的前缀组成的索引。举例来说，如果有一个 `{"a": 1, "b": 1, "c": 1, ..., "z": 1}` 索引，那么，实际上我们也可以使用 `{"a": 1}`、`{"a": 1, "b" : 1}`、`{"a": 1, "b": 1, "c": 1}` 等一系列索引。

注意，这些键的任意子集所组成的索引并不一定可用。例如，使用 `{"b": 1}` 或者 `{"a": 1, "c": 1}` 作为索引的查询是不会被优化的：只有能够使用索引前缀的查询才能从中受益。

5.1.3 $操作符如何使用索引

有一些查询完全无法使用索引，也有一些查询能够比其他查询更高效地使用索引。

本节讲述 MongoDB 对各种不同查询操作符的处理。

1. 低效率的操作符

有一些查询完全无法使用索引，比如 "$where" 查询和检查一个键是否存在的查询（{"*key*" : {"$exists" : true}}）。也有其他一些操作不能高效地使用索引。

如果 "x" 上有一个索引，查询那些不包含 "x" 键的文档可以使用这样的索引({"x" : {"$exists" : false}}。然而，在索引中，不存在的字段和 null 字段的存储方式是一样的，查询必须遍历每一个文档检查这个值是否真的为 null 还是根本不存在。如果使用稀疏索引（sparse index），就不能使用 {"$exists" : true}，也不能使用 {"$exists" : false}。

通常来说，取反的效率是比较低的。"$ne" 查询可以使用索引，但并不是很有效。因为必须要查看所有的索引条目，而不只是 "$ne" 指定的条目，不得不扫描整个索引。例如，这样的查询遍历的索引范围如下：

```
> db.example.find({"i" : {"$ne" : 3}}).explain()
{
    "cursor" : "BtreeCursor i_1 multi",
    ...,
    "indexBounds" : {
        "i" : [
            [
                {
                    "$minElement" : 1
                },
                3
            ],
            [
                3,
                {
                    "$maxElement" : 1
                }
            ]
        ]
    },
    ...
}
```

这个查询查找了所有小于 3 和大于 3 的索引条目。如果索引中值为 3 的条目非常多，那么这个查询的效率是很不错的，否则的话，这个查询就不得不检查几乎所有的索引条目。

"$not" 有时能够使用索引，但是通常它并不知道要如何使用索引。它能够对基本的范围（比如将 {"*key*" : {"$lt" : 7}} 变成 {"*key*" : {"$gte" : 7}}）和正则表达式进行反转。然而，大多数使用 "$not" 的查询都会退化为进行全表扫描。

"$nin" 就总是进行全表扫描。

如果需要快速执行一个这些类型的查询，可以试着找到另一个能够使用索引的语句，将其添加到查询中，这样就可以在 MongoDB 进行无索引匹配（non-indexed matching）时先将结果集的文档数量减到一个比较小的水平。

假如我们要找出所有没有 "birthday" 字段的用户。如果我们知道从 3 月 20 开始，程序会为每一个新用户添加生日字段，那么就可以只查询 3 月 20 之前创建的用户：

```
> db.users.find({"birthday" : {"$exists" : false}, "_id" : {"$lt" : march20Id}})
```

这个查询中的字段顺序无关紧要，MongoDB 会自动找出可以使用索引的字段，而无视查询中的字段顺序。

2. 范围

复合索引使 MongoDB 能够高效地执行拥有多个语句的查询。设计基于多个字段的索引时，应该将会用于精确匹配的字段（比如 "x" : "foo"）放在索引的前面，将用于范围匹配的字段（比如 "y" : {"$gt" : 3，"$lt" : 5}）放在最后。这样，查询就可以先使用第一个索引键进行精确匹配，然后再使用第二个索引范围在这个结果集内部进行搜索。假设要使用 {"age" : 1，"username" : 1} 索引查询特定年龄和用户名范围内的文档，可以精确指定索引边界值：

```
> db.users.find({"age" : 47,
... "username" : {"$gt" : "user5", "$lt" : "user8"}}).explain()
{
    "cursor" : "BtreeCursor age_1_username_1",
    "n" : 2788,
    "nscanned" : 2788,
    ...,
    "indexBounds" : {
        "age" : [
            [
                47,
                47
            ]
        ],
        "username" : [
            [
                "user5",
                "user8"
            ]
        ]
    },
    ...
}
```

这个查询会直接定位到 "age" 为 47 的索引条目，然后在其中搜索用户名介于

"user5" 和 "user8" 的条目。

反过来，假如使用 {"username" : 1, "age" : 1} 索引，这样就改变了查询计划（query plan），查询必须先找到介于 "user5" 和 "user8" 之间的所有用户，然后再从中挑选 "age" 等于 47 的用户。

```
> db.users.find({"age" : 47,
... "username" : {"$gt" : "user5", "$lt" : "user8"}}).explain()
{
    "cursor" : "BtreeCursor username_1_age_1",
    "n" : 2788,
    "nscanned" : 319499,
    ...,
    "indexBounds" : {
        "username" : [
            [
                "user5",
                "user8"
            ]
        ],
        "age" : [
            [
                47,
                47
            ]
        ]
    },
    "server" : "spock:27017"
}
```

本次查询中 MongoDB 扫描的索引条目数量是前一个查询的 10 倍！在一次查询中使用两个范围通常会导致低效的查询计划。

3. OR查询

写作本书时，MongoDB 在一次查询中只能使用一个索引。如果你在 {"x" : 1} 上有一个索引，在 {"y" : 1} 上也有一个索引，在 {"x" : 123, "y" : 456} 上进行查询时，MongoDB 会使用其中的一个索引，而不是两个一起用。"$or" 是个例外，"$or" 可以对每个子句都使用索引，因为 "$or" 实际上是执行两次查询然后将结果集合并。

```
> db.foo.find({"$or" : [{"x" : 123}, {"y" : 456}]}).explain()
{
     "clauses" : [
         {
             "cursor" : "BtreeCursor x_1",
             "isMultiKey" : false,
             "n" : 1,
             "nscannedObjects" : 1,
             "nscanned" : 1,
```

```
                "nscannedObjectsAllPlans" : 1,
                "nscannedAllPlans" : 1,
                "scanAndOrder" : false,
                "indexOnly" : false,
                "nYields" : 0,
                "nChunkSkips" : 0,
                "millis" : 0,
                "indexBounds" : {
                    "x" : [
                        [
                          123,
                          123
                        ]
                    ]
            }
        },
        {
            "cursor" : "BtreeCursor y_1",
            "isMultiKey" : false,
            "n" : 1,
            "nscannedObjects" : 1,
            "nscanned" : 1,
            "nscannedObjectsAllPlans" : 1,
            "nscannedAllPlans" : 1,
            "scanAndOrder" : false,
            "indexOnly" : false,
            "nYields" : 0,
            "nChunkSkips" : 0,
            "millis" : 0,
            "indexBounds" : {
                "y" : [
                    [
                        456,
                        456
                    ]
                ]
            }
        }
    ],
    "n" : 2,
    "nscannedObjects" : 2,
    "nscanned" : 2,
    "nscannedObjectsAllPlans" : 2,
    "nscannedAllPlans" : 2,
    "millis" : 0,
    "server" : "spock:27017"
}
```

可以看到，这次的 `explain()` 输出结果由两次独立的查询组成。通常来说，执行两次查询再将结果合并的效率不如单次查询高，因此，应该尽可能使用 `"$in"` 而不是 `"$or"`。

如果不得不使用 `"$or"`，记住，MongoDB 需要检查每次查询的结果集并且从中移

除重复的文档（有些文档可能会被多个 "$or" 子句匹配到）。

使用 "$in" 查询时无法控制返回文档的顺序（除非进行排序）。例如，使用 {"x" : [1，2，3]} 与使用 {"x" : [3，2，1]} 得到的文档顺序是相同的。

5.1.4　索引对象和数组

MongoDB 允许深入文档内部，对嵌套字段和数组建立索引。嵌套对象和数组字段可以与复合索引中的顶级字段一起使用，虽然它们比较特殊，但是大多数情况下与“正常”索引字段的行为是一致的。

1. 索引嵌套文档

可以在嵌套文档的键上建立索引，方式与正常的键一样。如果有这样一个集合，其中的第一个文档表示一个用户，可能需要使用嵌套文档来表示每个用户的位置：

```
{
    "username" : "sid",
    "loc" : {
        "ip" : "1.2.3.4",
        "city" : "Springfield",
        "state" : "NY"
    }
}
```

需要在 "loc" 的某一个子字段（比如 "loc.city"）上建立索引，以便提高这个字段的查询速度：

```
> db.users.ensureIndex({"loc.city" : 1})
```

可以用这种方式对任意深层次的字段建立索引，比如你可以在 "x.y.z.w.a.b.c" 上建立索引。

注意，对嵌套文档本身（"loc"）建立索引，与对嵌套文档的某个字段（"loc.city"）建立索引是不同的。对整个子文档建立索引，只会提高整个子文档的查询速度。在上面的例子中，只有在进行与子文档字段顺序完全匹配的子文档查询时（比如 db.users.find({"loc" : {"ip" : "123.456.789.000"， "city" : "Shelbyville"， "state" : "NY"}}})），查询优化器才会使用 "loc" 上的索引。无法对形如 db.users.find({"loc.city" : "Shelbyville"}) 的查询使用索引。

2. 索引数组

也可以对数组建立索引，这样就可以高效地搜索数组中的特定元素。

假如有一个博客文章的集合，其中每个文档表示一篇文章。每篇文章都有一个 "comments" 字段，这是一个数组，其中每个元素都是一个评论子文档。如果想要找出最近被评论次数最多的博客文章，可以在博客文章集合中嵌套的 "comments" 数组的 "date" 键上建立索引：

```
> db.blog.ensureIndex({"comments.date" : 1})
```

对数组建立索引，实际上是对数组的每一个元素建立一个索引条目，所以如果一篇文章有 20 条评论，那么它就拥有 20 个索引条目。因此数组索引的代价比单值索引高：对于单次插入、更新或者删除，每一个数组条目可能都需要更新（可能有上千个索引条目）。

与上一节中 "loc" 的例子不同，无法将整个数组作为一个实体建立索引：对数组建立索引，实际上是对数组中的每个元素建立索引，而不是对数组本身建立索引。

在数组上建立的索引并不包含任何位置信息：无法使用数组索引查找特定位置的数组元素，比如 "comments.4"。

少数特殊情况下，可以对某个特定的数组条目进行索引，比如：

```
> db.blog.ensureIndex({"comments.10.votes": 1})
```

然而，只有在精确匹配第 11 个数组元素时这个索引才有用（数组下标从 0 开始）。

一个索引中的数组字段最多只能有一个。这是为了避免在多键索引中索引条目爆炸性增长：每一对可能的元素都要被索引，这样导致每个文档拥有 n*m 个索引条目。假如有一个 {"x" : 1，"y" : 1} 上的索引：

```
> // x 是一个数组——这是合法的
> db.multi.insert({"x" : [1, 2, 3], "y" : 1})
>
> // y 是一个数组——这也是合法的
> db.multi.insert({"x" : 1, "y" : [4, 5, 6]})
>
> // x 和 y 都是数组——这是非法的！
> db.multi.insert({"x" : [1, 2, 3], "y" : [4, 5, 6]})
cannot index parallel arrays [y] [x]
```

如果 MongoDB 要为上面的最后一个例子创建索引，它必须要创建这么多索引条目：{"x" : 1，"y" : 4}、{"x" : 1，"y" : 5}、{"x" : 1，"y" : 6}、{"x" : 2，"y" : 4}、{"x" : 2，"y" : 5}，{"x" : 2，"y" : 6}、{"x" : 3，"y" : 4}、{"x" : 3，"y" : 5} 和 {"x" : 3，"y" : 6}。尽管这些数组只有 3 个元素。

3. 多键索引

对于某个索引的键，如果这个键在某个文档中是一个数组，那么这个索引就会被标记为多键索引（multikey index）。可以从 `explain()` 的输出中看到一个索引是否为多键索引：如果使用了多键索引，`"isMultikey"` 字段的值会是 `true`。索引只要被标记为多键索引，就无法再变成非多键索引了，即使这个字段为数组的所有文档都从集合中删除。要将多键索引恢复为非多键索引，唯一的方法就是删除再重建这个索引。

多键索引可能会比非多键索引慢一些。可能会有多个索引条目指向同一个文档，因此 MongoDB 在返回结果集时必须要先去除重复的内容。

5.1.5　索引基数

基数（cardinality）就是集合中某个字段拥有不同值的数量。有一些字段，比如 `"gender"` 或者 `"newsletter opt-out"`，可能只拥有两个可能的值，这种键的基数就是非常低的。另外一些字段，比如 `"username"` 或者 `"email"`，可能集合中的每个文档都拥有一个不同的值，这类键的基数是非常高的。当然也有一些介于两者之间的字段，比如 `"age"` 或者 `"zip code"`。

通常，一个字段的基数越高，这个键上的索引就越有用。这是因为索引能够迅速将搜索范围缩小到一个比较小的结果集。对于低基数的字段，索引通常无法排除掉大量可能的匹配。

假设我们在 `"gender"` 上有一个索引，需要查找名为 Susan 的女性用户。通过这个索引，只能将搜索空间缩小到大约 50%，然后要在每个单独的文档中查找 `"name"` 为 `"Susan"` 的用户。反过来，如果在 `"name"` 上建立索引，就能立即将结果集缩小到名为 `"Susan"` 的用户，这样的结果集非常小，然后就可以根据性别从中迅速地找到匹配的文档了。

一般说来，应该在基数比较高的键上建立索引，或者至少应该把基数较高的键放在复合索引的前面（低基数的键之前）。

5.2　使用explain()和hint()

从上面的内容可以看出，`explain()` 能够提供大量与查询相关的信息。对于速度比较慢的查询来说，这是最重要的诊断工具之一。通过查看一个查询的 `explain()` 输出信息，可以知道查询使用了哪个索引，以及是如何使用的。对于任意查询，都可以在最后添加一个 `explain()` 调用（与调用 `sort()` 或者 `limit()` 一样，不过

explain() 必须放在最后)。

最常见的 explain() 输出有两种类型：使用索引的查询和没有使用索引的查询。对于特殊类型的索引，生成的查询计划可能会有些许不同，但是大部分字段都是相似的。另外，分片返回的是多个 explain() 的聚合（第 13 章会介绍），因为查询会在多个服务器上执行。

不使用索引的查询的 exlpain() 是最基本的 explain() 类型。如果一个查询不使用索引，是因为它使用了 "BasicCursor"（基本游标）。反过来说，大部分使用索引的查询使用的是 BtreeCursor（某些特殊类型的索引，比如地理空间索引，使用的是它们自己类型的游标）。

对于使用了复合索引的查询，最简单情况下的 explain() 输出如下所示：

```
> db.users.find({"age" : 42}).explain()
{
    "cursor" : "BtreeCursor age_1_username_1",
    "isMultiKey" : false,
    "n" : 8332,
    "nscannedObjects" : 8332,
    "nscanned" : 8332,
    "nscannedObjectsAllPlans" : 8332,
    "nscannedAllPlans" : 8332,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 91,
    "indexBounds" : {
        "age" : [
            [
                42,
                42
            ]
        ],
        "username" : [
            [
                {
                    "$minElement" : 1
                },
                {
                    "$maxElement" : 1
                }
            ]
        ]
    },
    "server" : "ubuntu:27017"
}
```

从输出信息中可以看到它使用的索引是 age_1_username_1。"millis" 表明了这

个查询的执行速度，时间是从服务器收到请求开始一直到发出响应为止。然而，这个数值不一定真的是你希望看到的值。如果 MongoDB 尝试了多个查询计划，那么 `"millis"` 显示的是这些查询计划花费的总时间，而不是最优查询计划所花的时间。

接下来是实际返回的文档数量：`"n"`。它无法反映出 MongoDB 在执行这个查询的过程中所做的工作：搜索了多少索引条目和文档。索引条目是使用 `"nscanned"` 描述的。`"nscannedObjects"` 字段的值就是所扫描的文档数量。最后，如果要对结果集进行排序，而 MongoDB 无法对排序使用索引，那么 `"scanAndOrder"` 的值就会是 `true`。也就是说，MongoDB 不得不在内存中对结果进行排序，这是非常慢的，而且结果集的数量要比较小。

现在你已经知道这些基础知识了，接下来依次详细介绍这些字段。

- `"cursor" : "BtreeCursor age_1_username_1"`
  `BtreeCursor` 表示本次查询使用了索引，具体来说，是使用了 `"age"` 和 `"username"` 上的索引 `{"age" : 1，"username" : 1}`。如果查询要对结果进行逆序遍历，或者是使用了多键索引，就可以在这个字段中看到 `"reverse"` 和 `"multi"` 这样的值。

- `"isMultiKey" : false`
  用于说明本次查询是否使用了多键索引（详见 5.1.4 节）。

- `"n" : 8332`
  本次查询返回的文档数量。

- `"nscannedObjects" : 8332`
  这是 MongoDB 按照索引指针去磁盘上查找实际文档的次数。如果查询包含的查询条件不是索引的一部分，或者说要求返回不在索引内的字段，MongoDB 就必须依次查找每个索引条目指向的文档。

- `"nscanned" : 8332`
  如果有使用索引，那么这个数字就是查找过的索引条目数量。如果本次查询是一次全表扫描，那么这个数字就表示检查过的文档数量。

- `"scanAndOrder" : false`
  MongoDB 是否在内存中对结果集进行了排序。

- `"indexOnly" : false`
  MongoDB 是否只使用索引就能完成此次查询（详见“覆盖索引”部分）。

在本例中，MongoDB 只使用索引就找到了全部的匹配文档，从 "nscanned" 和 "n" 相等就可以看出来。然而，本次查询要求返回匹配文档中的所有字段，而索引只包含 "age" 和 "username" 两个字段。如果将本次查询修改为（{"_id" : 0， "age" : 1， "username" : 1}），那么本次查询就可以被索引覆盖了，"indexOnly" 的值就会是 true。

- "nYields" : 0

  为了让写入请求能够顺利执行，本次查询暂停的次数。如果有写入请求需要处理，查询会周期性地释放它们的锁，以便写入能够顺利执行。然而，在本次查询中，没有写入请求，因为查询没有暂停过。

- "millis" : 91

  数据库执行本次查询所耗费的毫秒数。这个数字越小，说明查询效率越高。

- "indexBounds" : {...}

  这个字段描述了索引的使用情况，给出了索引的遍历范围。由于查询中的第一个语句是精确匹配，因此索引只需要查找 42 这个值就可以了。本次查询没有指定第二个索引键，因此这个索引键上没有限制，数据库会在 "age" 为 42 的条目中将用户名介于负无穷（"$minElement" : 1）和正无穷（"$maxElement" : 1）的条目都找出来。

再来看一个稍微复杂点的例子：假如有一个 {"user name" : 1， "age" : 1} 上的索引和一个 {"age" : 1， "username" : 1} 上的索引。同时查询 "username" 和 "age" 时，会发生什么情况？呃，这取决于具体的查询：

```
> db.c.find({age : {$gt : 10}, username : "sally"}).explain()
{
    "cursor" : "BtreeCursor username_1_age_1",
    "indexBounds" : [
        [
            {
                "username" : "sally",
                "age" : 10
            },
            {
                "username" : "sally",
                "age" : 1.7976931348623157e+308
            }
        ]
    ],
    "nscanned" : 13,
    "nscannedObjects" : 13,
    "n" : 13,
    "millis" : 5
}
```

由于在要在 "username" 上执行精确匹配，在 "age" 上进行范围查询，因此，数据库选择使用 {"username" : 1, "age" : 1} 索引，这与查询语句的顺序相反。另一方面来说，如果需要对 "age" 精确匹配而对 "username" 进行范围查询，MongoDB 就会使用另一个索引：

```
> db.c.find({"age" : 14, "username" : /.*/}).explain()
{
    "cursor" : "BtreeCursor age_1_username_1 multi",
    "indexBounds" : [
        [
            {
                "age" : 14,
                "username" : ""
            },
            {
                "age" : 14,
                "username" : {
                }
            }
        ],
        [
            {
                "age" : 14,
                "username" : /.*/
            },
            {
                "age" : 14,
                "username" : /.*/
            }
        ]
    ],
    "nscanned" : 2,
    "nscannedObjects" : 2,
    "n" : 2,
    "millis" : 2
}
```

如果发现 MongoDB 使用的索引与自己希望它使用的索引不一致，可以使用 hint() 强制 MongoDB 使用特定的索引。例如，如果希望 MongoDB 在上个例子的查询中使用 {"username" : 1, "age" : 1} 索引，可以这么做：

```
> db.c.find({"age" : 14, "username" : /.*/}).hint({"username" : 1, "age" : 1})
```

如果查询没有使用你希望它使用的索引，于是你使用 hint 强制 MongoDB 使用某个索引，那么应该在应用程序部署之前在所指定的索引上执行 explain()。如果强制 MongoDB 在某个查询上使用索引，而这个查询不知道如何使用这个索引，这样会导致查询效率降低，还不如不使用索引来得快。

查询优化器

MongoDB 的查询优化器与其他数据库稍有不同。基本来说，如果一个索引能够精确匹配一个查询（要查询 "x"，刚好在 "x" 上有一个索引），那么查询优化器就会使用这个索引。不然的话，可能会有几个索引都适合你的查询。MongoDB 会从这些可能的索引子集中为每次查询计划选择一个，这些查询计划是并行执行的。最早返回 100 个结果的就是胜者，其他的查询计划就会被中止。

这个查询计划会被缓存，这个查询接下来都会使用它，直到集合数据发生了比较大的变动。如果在最初的计划评估之后集合发生了比较大的数据变动，查询优化器就会重新挑选可行的查询计划。建立索引时，或者是每执行 1000 次查询之后，查询优化器都会重新评估查询计划。

explain() 输出信息里的 "allPlans" 字段显示了本次查询尝试过的每个查询计划。

5.3 何时不应该使用索引

提取较小的子数据集时，索引非常高效。也有一些查询不使用索引会更快。结果集在原集合中所占的比例越大，索引的速度就越慢，因为使用索引需要进行两次查找：一次是查找索引条目，一次是根据索引指针去查找相应的文档。而全表扫描只需要进行一次查找：查找文档。在最坏的情况下（返回集合内的所有文档），使用索引进行的查找次数会是全表扫描的两倍，效率会明显比全表扫描低很多。

可惜，并没有一个严格的规则可以告诉我们，如何根据数据大小、索引大小、文档大小以及结果集的平均大小来判断什么时候索引很有用，什么时候索引会降低查询速度（如表 5-1 所示）。一般来说，如果查询需要返回集合内 30% 的文档（或者更多），那就应该对索引和全表扫描的速度进行比较。然而，这个数字可能会在 2% ~ 60% 之间变动。

表5-1 影响索引效率的属性

索引通常适用的情况	全表扫描通常适用的情况
集合较大	集合较小
文档较大	文档较小
选择性查询	非选择性查询

假如我们有一个收集统计信息的分析系统。应用程序要根据给定账户去系统中查询所有文档，根据从初始一直到一小时之前的数据生成图表：

```
> db.entries.find({"created_at" : {"$lt" : hourAgo}})
```

我们在 "created_at" 上创建索引以提高查询速度。

最初运行时，结果集非常小，可以立即返回。几个星期过去以后，数据开始多起来了，一个月之后，这个查询耗费的时间越来越长。

对于大部分应用程序来说，这很可能就是那个“错误的”查询：真的需要在查询中返回数据集中的大部分内容吗？大部分应用程序（尤其是拥有非常大的数据集的应用程序）都不需要。然而，也有一些合理的情况，可能需要得到大部分或者全部的数据：也许需要将这些数据导出到报表系统，或者是放在批量任务中。在这些情况下，应该尽可能快地返回数据集中的内容。

可以用 {"$natural" : 1} 强制数据库做全表扫描。6.1 节会介绍 $natural，它可以指定文档按照磁盘上的顺序排列。特别地，$natural 可以强制 MongoDB 做全表扫描：

```
> db.entries.Ǹnd({"created_at" : {"$lt" : hourAgo}}).hint({"$natural" : 1})
```

使用 "$natural" 排序有一个副作用：返回的结果是按照磁盘上的顺序排列的。对于一个活跃的集合来说，这是没有意义的：随着文档体积的增加或者缩小，文档会在磁盘上进行移动，新的文档会被写入到这些文档留下的空白位置。但是，对于只需要进行插入的工作来说，如果要得到最新的（或者最早的）文档，使用 $natural 就非常有用了。

5.4 索引类型

创建索引时可以指定一些选项，使用不同选项建立的索引会有不同的行为。接下来的小节会介绍常见的索引变种，更高级的索引类型和特殊选项会在下一章介绍。

5.4.1 唯一索引

唯一索引可以确保集合的每一个文档的指定键都有唯一值。例如，如果想保证同不文档的 "username" 键拥有不同的值，创建一个唯一索引就好了：

```
> db.users.ensureIndex({"username" : 1}, {"unique" : true})
```

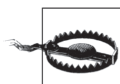
如果试图向上面的集合中插入如下文档：

```
> db.users.insert({username: "bob"})
> db.users.insert({username: "bob"})
E11000 duplicate key error index: test.users.$username_1 dup key: { : "bob" }
```

如果检查这个集合，会发现只有第一个 "bob" 被保存进来了。发现有重复的键时抛

出异常会影响效率，所以可以使用唯一索引来应对偶尔可能会出现的键重复问题，而不是在运行时对重复的键进行过滤。

有一个唯一索引可能你已经比较熟悉了，就是 "_id" 索引，这个索引会在创建集合时自动创建。这就是一个正常的唯一索引（但它不能被删除，而其他唯一索引是可以删除的）。



如果一个文档没有对应的键，索引会将其作为 null 存储。所以，如果对某个键建立了唯一索引，但插入了多个缺少该索引键的文档，由于集合已经存在一个该索引键的值为 null 的文档而导致插入失败。5.4.2 节会详细介绍相关内容。

有些情况下，一个值可能无法被索引。索引储桶（index bucket）的大小是有限制的，如果某个索引条目超出了它的限制，那么这个条目就不会包含在索引里。这样会造成一些困惑，因为使用这个索引进行查询时会有一个文档凭空消失不见了。所有的字段都必须小于 1024 字节，才能包含到索引里。如果一个文档的字段由于太大不能包含在索引里，MongoDB 不会返回任何错误或者警告。也就是说，超出 1 KB 大小的键不会受到唯一索引的约束：可以插入多个同样的 1 KB 长的字符串。

1. 复合唯一索引

也可以创建复合的唯一索引。创建复合唯一索引时，单个键的值可以相同，但所有键的组合值必须是唯一的。

例如，如果有一个 {"username" : 1, "age" : 1} 上的唯一索引，下面的插入是合法的：

```
> db.users.insert({"username" : "bob"})
> db.users.insert({"username" : "bob", "age" : 23})
> db.users.insert({"username" : "fred", "age" : 23})
```

然而，如果试图再次插入这三个文档中的任意一个，都会导致键重复异常。

GirdFS 是 MongoDB 中存储大文件的标准方式（详见 6.5 节），其中就用到了复合唯一索引。存储文件内容的集合有一个 {"files_id" : 1, "n" : 1} 上的复合唯一索引，因此文档的某一部分看起来可能会是下面这个样子：

```
{"files_id" : ObjectId("4b23c3ca7525f35f94b60a2d"), "n" : 1}
{"files_id" : ObjectId("4b23c3ca7525f35f94b60a2d"), "n" : 2}
{"files_id" : ObjectId("4b23c3ca7525f35f94b60a2d"), "n" : 3}
{"files_id" : ObjectId("4b23c3ca7525f35f94b60a2d"), "n" : 4}
```

注意，所有 "files_id" 的值都相同，但是 "n" 的值不同。

2. 去除重复

在已有的集合上创建唯一索引时可能会失败，因为集合中可能已经存在重复值了：

```
> db.users.ensureIndex({"age" : 1}, {"unique" : true})
E11000 duplicate key error index: test.users.$age_1 dup key: { : 12 }
```

通常需要先对已有的数据进行处理（可以使用聚合框架），找出重复的数据，想办法处理。

在极少数情况下，可能希望直接删除重复的值。创建索引时使用 `"dropDups"` 选项，如果遇到重复的值，第一个会被保留，之后的重复文档都会被删除。

```
> db.people.ensureIndex({"username" : 1}, {"unique" : true, "dropDups" : true})
```

`"dropDups"` 会强制性建立唯一索引，但是这个方式太粗暴了：你无法控制哪些文档被保留哪些文档被删除（如果有文档被删除的话，MongoDB 也不会给出提示说哪些文档被删除了）。对于比较重要的数据，千万不要使用 `"dropDups"`。

5.4.2 稀疏索引

前面的小节已经讲过，唯一索引会把 `null` 看做值，所以无法将多个缺少唯一索引中的键的文档插入到集合中。然而，在有些情况下，你可能希望唯一索引只对包含相应键的文档生效。如果有一个可能存在也可能不存在的字段，但是当它存在时，它必须是唯一的，这时就可以将 `unique` 和 `sparse` 选项组合在一起使用。



MongoDB 中的稀疏索引（sparse index）与关系型数据库中的稀疏索引是完全不同的概念。基本上来说，MongoDB 中的稀疏索引只是不需要将每个文档都作为索引条目。

使用 `sparse` 选项就可以创建稀疏索引。例如，如果有一个可选的 email 地址字段，但是，如果提供了这个字段，那么它的值必须是唯一的：

```
> db.ensureIndex({"email" : 1}, {"unique" : true, "sparse" : true})
```

稀疏索引不必是唯一的。只要去掉 `unique` 选项，就可以创建一个非唯一的稀疏索引。

根据是否使用稀疏索引，同一个查询的返回结果可能会不同。假如有这样一个集合，其中的大部分文档都有一个 `"x"` 字段，但是有些没有：

```
> db.foo.find()
{ "_id" : 0 }
```

```
{ "_id" : 1, "x" : 1 }
{ "_id" : 2, "x" : 2 }
{ "_id" : 3, "x" : 3 }
```

当在 "x" 上执行查询时，它会返回相匹配的文档：

```
> db.foo.find({"x" : {"$ne" : 2}})
{ "_id" : 0 }
{ "_id" : 1, "x" : 1 }
{ "_id" : 3, "x" : 3 }
```

如果在 "x" 上创建一个稀疏索引，"_id" 为 0 的文档就不会包含在索引中。如果再次在 "x" 上查询，MongoDB 就会使用这个稀疏索引，{"_id" : 0} 的这个文档就不会被返回了：

```
> db.foo.find({"x" : {"$ne" : 2}})
{ "_id" : 1, "x" : 1 }
{ "_id" : 3, "x" : 3 }
```

如果需要得到那些不包含 "x" 字段的文档，可以使用 hint() 强制进行全表扫描。

5.5 索引管理

如前面的小节所述，可以使用 ensuerIndex 函数创建新的索引。对于一个集合，每个索引只需要创建一次。如果重复创建相同的索引，是没有任何作用的。

所有的数据库索引信息都存储在 system.indexes 集合中。这是一个保留集合，不能在其中插入或者删除文档。只能通过 ensureIndex 或者 dropIndexes 对其进行操作。

创建一个索引之后，就可以在 system.indexes 中看到它的元信息。可以执行 db.collectionName.getIndexes() 来查看给定集合上的所有索引信息：

```
> db.foo.getIndexes()
[
    {
        "v" : 1,
        "key" : {
            "_id" : 1
        },
        "ns" : "test.foo",
        "name" : "_id_"
    },
    {
        "v" : 1,
        "key" : {
            "y" : 1
```

```
        },
        "ns" : "test.foo",
        "name" : "y_1"
    },
    {
        "v" : 1,
        "key" : {
            "x" : 1,
            "y" : 1
        },
        "ns" : "test.foo",
        "name" : "x_1_y_1"
    }
]
```

这里面最重要的字段是 `"key"` 和 `"name"`。这里的键可以用在 `hint`、`max`、`min` 以及其他所有需要指定索引的地方。在这里，索引的顺序很重要：`{"x" : 1, "y" : 1}` 上的索引与 `{"y" : 1, "x" : 1}` 上的索引不同。对于很多的索引操作（比如 `dropIndex`），这里的索引名称都可以被当作标识符使用。但是这里不会指明索引是否是多键索引。

`"v"` 字段只在内部使用，用于标识索引版本。如果你的索引不包含 `"v" : 1` 这样的字段，说明你的索引是以一种效率比较低的旧方式存储的。将 MongoDB 升级到至少 2.0 版本，删除并重建这些索引，就可以把索引的存储方式升级到新的格式了。

5.5.1　标识索引

集合中的每一个索引都有一个名称，用于唯一标识这个索引，也可以用于服务器端来删除或者操作索引。索引名称的默认形式是 *`keyname1_dir1_keyname2_dir2_..._keynameN_dirN`*，其中 *`keynameX`* 是索引的键，*`dirX`* 是索引的方向（1 或者 –1）。如果索引中包含两个以上的键，这种命名方式就显得比较笨重了，好在可以在 `ensureIndex` 中指定索引的名称：

```
> db.foo.ensureIndex({"a" : 1, "b" : 1, "c" : 1, ..., "z" : 1},
... {"name" : "alphabet"})
```

索引名称的长度是有限制的，所以新建复杂索引时可能需要自定义索引名称。调用 `getLastError` 就可以知道索引是否成功创建，或者失败的原因。

5.5.2　修改索引

随着应用不断增长变化，你会发现数据或者查询已经发生了改变，原来的索引也不那么好用了。这时可以使用 `dropIndex` 命令删除不再需要的索引：

```
> db.people.dropIndex("x_1_y_1")
{ "nIndexesWas" : 3, "ok" : 1 }
```

用索引描述信息里 "name" 字段的值来指定需要删除的索引。

新建索引是一件既费时又浪费资源的事情。默认情况下，MongoDB 会尽可能快地创建索引，阻塞所有对数据库的读请求和写请求，一直到索引创建完成。如果希望数据库在创建索引的同时仍然能够处理读写请求，可以在创建索引时指定 background 选项。这样在创建索引时，如果有新的数据库请求需要处理，创建索引的过程就会暂停一下，但是仍然会对应用程序性能有比较大的影响（12.4.8 节会详细介绍）。后台创建索引比前台创建索引慢得多。

在已有的文档上创建索引会比新创建索引再插入文档快一点。

第 18 章会更详细地介绍实际创建索引。

第6章 特殊的索引和集合

本章介绍 MongoDB 中一些特殊的集合和索引类型，包括：

- 用于类队列数据的固定集合（capped collection）；
- 用于缓存的 TTL 索引；
- 用于简单字符串搜索的全文本索引；
- 用于二维平面和球体空间的地理空间索引；
- 用于存储大文件的 GridFS。

6.1 固定集合

MongoDB 中的“普通”集合是动态创建的，而且可以自动增长以容纳更多的数据。MongoDB 中还有另一种不同类型的集合，叫做*固定集合*，固定集合需要事先创建好，而且它的大小是固定的（如图 6-1 所示）。说到固定大小的集合，有一个很有趣的问题：向一个已经满了的固定集合中插入数据会怎么样？答案是，固定集合的行为类似于循环队列。如果已经没有空间了，最老的文档会被删除以释放空间，新插入的文档会占据这块空间（如图 6-2 所示）。也就是说，当固定集合被占满时，如果再插入新文档，固定集合会自动将最老的文档从集合中删除。

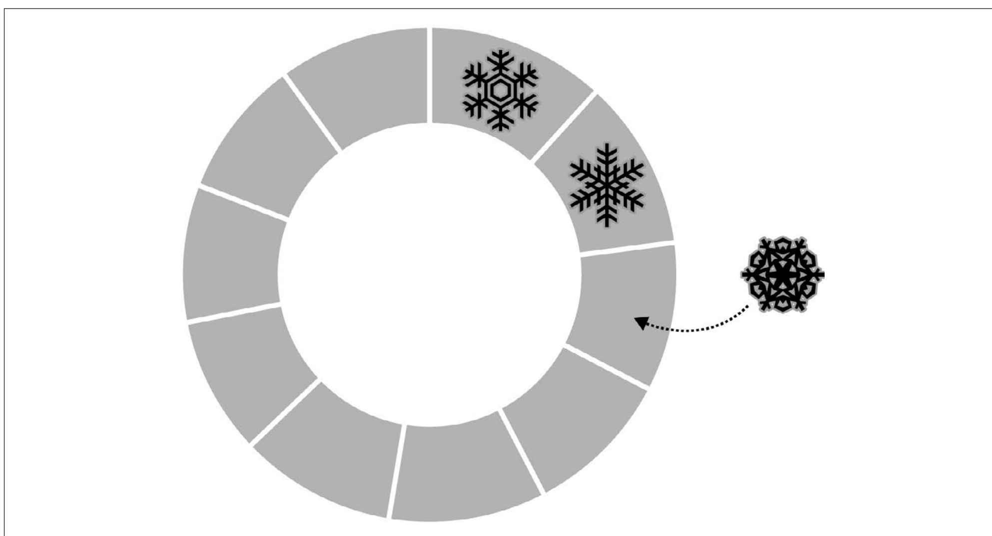


图 6-1：新文档被插入到队列末尾

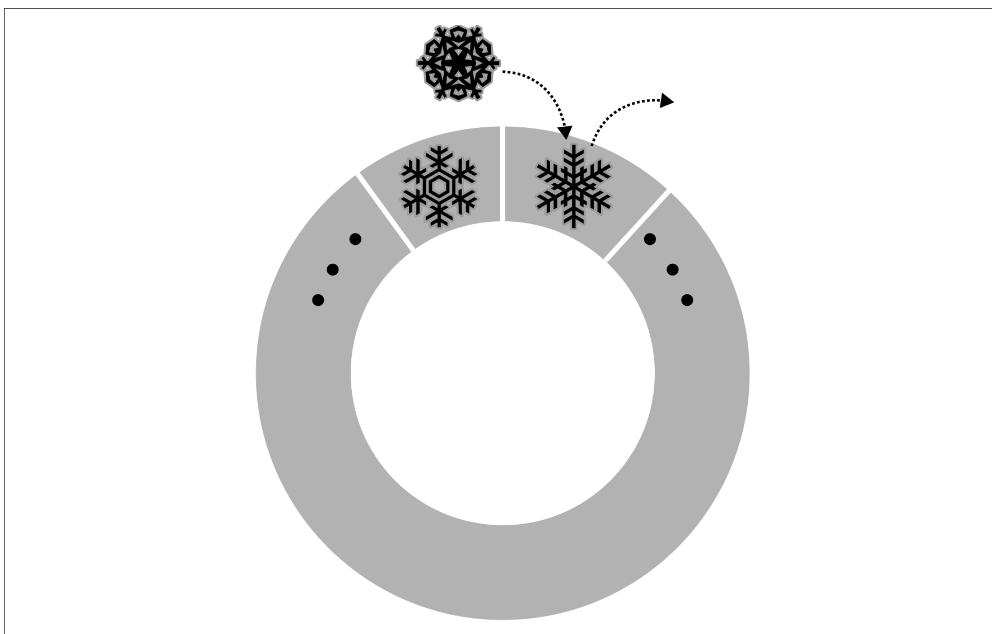


图 6-2：如果队列已经被占满，那么最老的文档会被之后插入的新文档覆盖

固定集合的访问模式与 MongoDB 中的大部分集合不同：数据被顺序写入磁盘上的固定空间。因此它们在碟式磁盘（spinning disk）上的写入速度非常快，尤其是集合拥有专用磁盘时（这样就不会因为其他集合的一些随机性的写操作而“中断”）。



固定集合不能被分片。

固定集合可以用于记录日志，尽管它们不够灵活。虽然可以在创建时指定集合大小，但无法控制什么时候数据会被覆盖。

6.1.1 创建固定集合

不同于普通集合，固定集合必须在使用之前先显式创建。可以使用 `create` 命令创建固定集合。在 shell 中，可以使用 `createCollection` 函数：

```
> db.createCollection("my_collection", {"capped" : true, "size" : 100000});
{ "ok" : true }
```

上面的命令创建了一个名为 `my_collection` 大小为 100 000 字节的固定集合。

除了大小，`createCollection` 还能够指定固定集合中文档的数量：

```
> db.createCollection("my_collection2",
... {"capped" : true, "size" : 100000, "max" : 100});
{ "ok" : true }
```

可以使用这种方式来保存最新的 10 则新闻，或者是将每个用户的文档数量限制为 1000。

固定集合创建之后，就不能改变了（如果需要修改固定集合的属性，只能将它删除之后再重建）。因此，在创建大的固定集合之前应该仔细想清楚它的大小。



为固定集合指定文档数量限制时，必须同时指定固定集合的大小。不管先达到哪一个限制，之后插入的新文档就会把最老的文档挤出集合：固定集合的文档数量不能超过文档数量限制，固定集合的大小也不能超过大小限制。

创建固定集合时还有另一个选项，可以将已有的某个常规集合转换为固定集合，可以使用 `convertToCapped` 命令实现。下面的例子将 `test` 集合转换为一个大小为 10 000 字节的固定集合：

```
> db.runCommand({"convertToCapped" : "test", "size" : 10000});
{ "ok" : true }
```

无法将固定集合转换为非固定集合（只能将其删除）。

6.1.2 自然排序

对固定集合可以进行一种特殊的排序，称为*自然排序*（natural sort）。自然排序返回结果集中文档的顺序就是文档在磁盘上的顺序（如图 6-3 所示）。

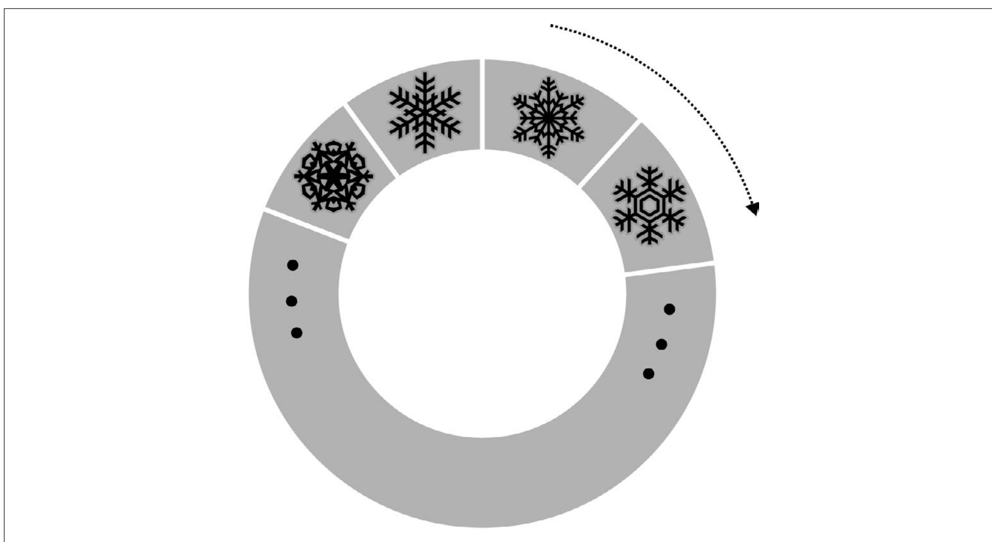


图 6-3：使用 {"$natural" : 1} 进行排序

对大多数集合来说，自然排序的意义不大，因为文档的位置经常变动。但是，固定集合中的文档是按照文档被插入的顺序保存的，自然顺序就是文档的插入顺序。因此，自然排序得到的文档是从旧到新排列的。当然也可以按照从新到旧的顺序排列（如图 6-4 所示）。

```
> db.my_collection.find().sort({"$natural" : -1})
```

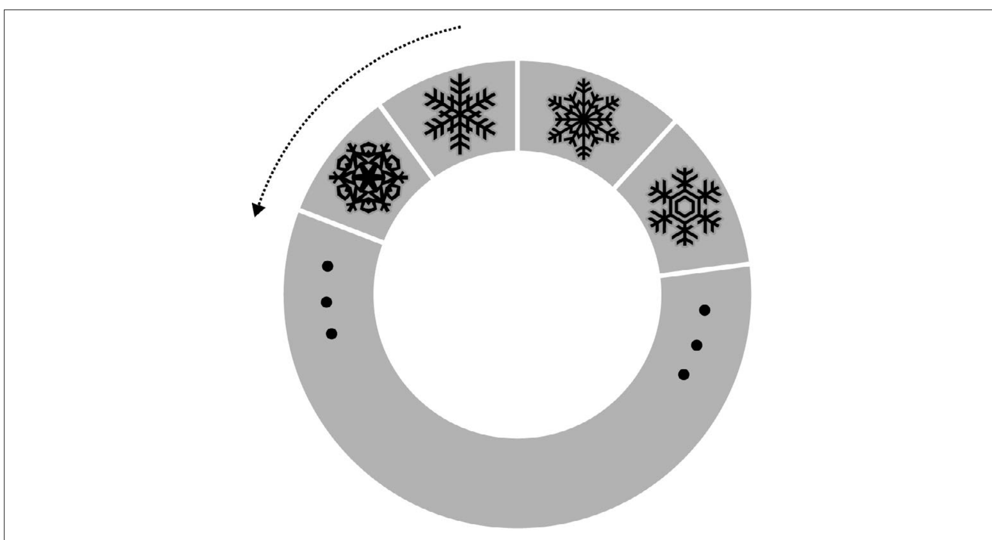


图 6-4：使用 {"$natural" : -1} 进行排序

6.1.3 循环游标

循环游标（tailable cursor）是一种特殊的游标，当循环游标的结果集被取光后，游标不会被关闭。循环游标的灵感来自 `tail -f` 命令（循环游标跟这个命令有点儿相似），会尽可能久地持续提取输出结果。由于循环游标在结果集取光之后不会被关闭，因此，当有新文档插入到集合中时，循环游标会继续取到结果。由于普通集合并不维护文档的插入顺序，所以循环游标只能用在固定集合上。

循环游标通常用于当文档被插入到“工作队列”（其实就是个固定集合）时对新插入的文档进行处理。如果超过 10 分钟没有新的结果，循环游标就会被释放，因此，当游标被关闭时自动重新执行查询是非常重要的。下面是一个在 PHP 中使用循环游标的例子（不能在 mongo shell 中使用循环游标）：

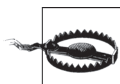
```
$cursor = $collection->find()->tailable();

while (true) {
    if (!$cursor->hasNext()) {
        if ($cursor->dead()) {
            break;
        }
        sleep(1);
    }
    else {
        while ($cursor->hasNext()) {
            do_stuff($cursor->getNext());
        }
    }
}
```

这个游标会不断对查询结果进行处理，或者是等待新的查询结果，直到游标被关闭（超过 10 分钟没有新的结果或者人为中止查询操作）。

6.1.4 没有_id索引的集合

默认情况下，每个集合都有一个 `"_id"` 索引。但是，如果在调用 `createCollection` 创建集合时指定 `autoIndexId` 选项为 `false`，创建集合时就不会自动在 `"_id"` 上创建索引。实践中不建议这么使用，但是对于只有插入操作的集合来说，这确实可以带来速度的稍许提升。



如果创建了一个没有 `"_id"` 索引的集合，那就永远都不能复制它所在的 mongod 了。复制操作要求每个集合上都要有 `"_id"` 索引（对于复制操作，能够唯一标识集合中的每一个文档是非常重要的）。

在 2.2 版本之前，固定集合默认是没有 `"_id"` 索引的，除非显式地将 `autoIndexId`

置为 true。如果正在使用旧版的固定集合，要确保你的应用程序能够填充 "_id" 字段（大多数驱动程序会自动填充 "_id" 字段），然后使用 ensureIndex 命令创建 "_id" 索引。

记住，"_id" 索引必须是唯一索引。不同于其他索引，"_id" 索引一经创建就无法删除了，因此在生产环境中创建索引之前先自己实践一下是非常重要的。所以创建 "_id" 索引必须一次成功！如果创建的 "_id" 索引不合规范，就只能删除集合再重建了。

6.2 TTL索引

上一节已经讲过，对于固定集合中的内容何时被覆盖，你只拥有非常有限的控制权限。如果需要更加灵活的老化移出系统（age-out system），可以使用 TTL 索引（time-to-live index，具有生命周期的索引），这种索引允许为每一个文档设置一个超时时间。一个文档到达预设置的老化程度之后就会被删除。这种类型的索引对于缓存问题（比如会话的保存）非常有用。

在 ensureIndex 中指定 expireAfterSecs 选项就可以创建一个 TTL 索引：

```
> // 超时时间为 24 小时
> db.foo.ensureIndex({"lastUpdated" : 1}, {"expireAfterSecs" : 60*60*24})
```

这样就在 "lastUpdated" 字段上建立了一个 TTL 索引。如果一个文档的 "lastUpdated" 字段存在并且它的值是日期类型，当服务器时间比文档的 "lastUpdated" 字段的时间晚 expireAfterSecs 秒时，文档就会被删除。

为了防止活跃的会话被删除，可以在会话上有活动发生时将 "lastUpdated" 字段的值更新为当前时间。只要 "lastUpdated" 的时间距离当前时间达到 24 小时，相应的文档就会被删除。

MongoDB 每分钟对 TTL 索引进行一次清理，所以不应该依赖以秒为单位的时间保证索引的存活状态。可以使用 collMod 命令修改 expireAfterSecs 的值：

```
> db.runCommand({"collMod" : "someapp.cache", "expireAfterSecs" : 3600})
```

在一个给定的集合上可以有多个 TTL 索引。TTL 索引不能是复合索引，但是可以像“普通”索引一样用来优化排序和查询。

6.3 全文本索引

MongoDB 有一个特殊类型的索引用于在文档中搜索文本。前面几章都是使用精确

匹配和正则表达式来查询字符串，但是这些技术有一些限制。使用正则表达式搜索大块文本的速度非常慢，而且无法处理语言的理解问题（比如 entry 与 entries 应该算是匹配的）。使用全文本索引可以非常快地进行文本搜索，就如同内置了多种语言分词机制的支持一样。

创建任何一种索引的开销都比较大，而创建全文本索引的成本更高。在一个操作频繁的集合上创建全文本索引可能会导致 MongoDB 过载，所以应该是离线状态下创建全文本索引，或者是在对性能没要求时。创建全文本索引时要特别小心谨慎，内存可能会不够用（除非你有 SSD）。第 18 章会介绍如何在创建索引时将对应用程序的影响降至最低。

全文本索引也会导致比“普通”索引更严重的性能问题，因为所有字符串都需要被分解、分词，并且保存到一些地方。因此，可能会发现拥有全文本索引的集合的写入性能比其他集合要差。全文本索引也会降低分片时的数据迁移速度：将数据迁移到其他分片时，所有文本都需要重新进行索引。

写作本书时，全文本索引仍然只是一个处于“试验阶段”的功能，所以需要专门启用这个功能才能进行使用。启动 MongoDB 时指定 `--setParameter textSearchEnabled=true` 选项，或者在运行时执行 `setParameter` 命令，都可以启用全文本索引：

```
> db.adminCommand({"setParameter" : 1, "textSearchEnabled" : true})
```

假如我们使用这个非官方的 Hacker News JSON API（http://api.ihackernews.com）将最近的一些文章加载到了 MongoDB 中。

为了进行文本搜索，首先需要创建一个 `"text"` 索引：

```
> db.hn.ensureIndex({"title" : "text"})
```

现在，必须通过 `text` 命令才能使用这个索引（写作本书时，全文本索引还不能用在“普通”查询中）：

```
test> db.runCommand({"text" : "hn", "search" : "ask hn"})
{
    "queryDebugString" : "ask|hn||||||",
    "language" : "english",
    "results" : [
        {
            "score" : 2.25,
            "obj" : {
                "_id" : ObjectId("50dcab296803fa7e4f000011"),
                "title" : "Ask HN: Most valuable skills you have?",
                "url" : "/comments/4974230",
```

```
                "id" : 4974230,
                "commentCount" : 37,
                "points" : 31,
                "postedAgo" : "2 hours ago",
                "postedBy" : "bavidar"
            }
        },
        {
            "score" : 0.5625,
            "obj" : {
                "_id" : ObjectId("50dcab296803fa7e4f000001"),
                "title" : "Show HN: How I turned an old book...",
                "url" : "http://www.howacarworks.com/about",
                "id" : 4974055,
                "commentCount" : 44,
                "points" : 95,
                "postedAgo" : "2 hours ago",
                "postedBy" : "AlexMuir"
            }
        },
        {
            "score" : 0.5555555555555556,
            "obj" : {
                 "_id" : ObjectId("50dcab296803fa7e4f000010"),
                 "title" : "Show HN: ShotBlocker - iOS Screenshot detector...",
                 "url" : "https://github.com/clayallsopp/ShotBlocker",
                 "id" : 4973909,
                 "commentCount" : 10,
                 "points" : 17,
                 "postedAgo" : "3 hours ago",
                 "postedBy" : "10char"
            }
        }
    ],
    "stats" : {
        "nscanned" : 4,
        "nscannedObjects" : 0,
        "n" : 3,
        "timeMicros" : 89
    },
    "ok" : 1
}
```

匹配到的文档是按照相关性降序排列的："Ask HN" 位于第一位，然后是两个部分匹配的文档。每个对象前面的 "score" 字段描述了每个结果与查询的匹配程度。

如你所见，这个搜索是不区分大小写不的，至少对于 [a-zA-Z] 这些字符是这样。全文本索引会使用 toLower 将单词变为小写，但这是与本地化相关的，所以某些语言的用户可能会发现 MongoDB 会不可预测性地变得区分大小写，这取决于 toLower 在不同字符集上的行为。MongoDB 一直在努力提高对不同字符集的支持。

全文本索引只会对字符串数据进行索引：其他的数据类型会被忽略，不会包含在索

引中。一个集合上最多只能有一个全文本索引，但是全文本索引可以包含多个字段：

```
> db.blobs.ensureIndex({"title" : "text", "desc" : "text", "author" : "text"})
```

与“普通”的多键索引不同，全文本索引中的字段顺序不重要：每个字段都被同等对待。可以为每个字段指定不同的权重来控制不同字段的相对重要性：

```
> db.hn.ensureIndex({"title" : "text", "desc" : "text", "author" : "text"},
... {"weights" : {"title" : 3, "author" : 2}})
```

默认的权重是 1，权重的范围可以是 1~1 000 000 000。使用上面的代码设置权重之后，`"title"` 字段成为其中最重要的字段，`"author"` 其次，最后是 `"desc"`（没有指定，因此它的权重是默认值 1）。

索引一经创建，就不能改变字段的权重了（除非删除索引再重建），所以在生产环境中创建索引之前应该先在测试数据集上实际操作一下。

对于某些集合，我们可能并不知道每个文档所包含的字段。可以使用 `"$**"` 在文档的所有字符串字段上创建全文本索引：这不仅会对顶级的字符串字段建立索引，也会搜索嵌套文档和数组中的字符串字段：

```
> db.blobs.ensureIndex({"$**" : "text"})
```

也可以为 `"$**"` 设置权重：

```
> db.hn.ensureIndex({"whatever" : "text"},
... {"weights" : {"title" : 3, "author" : 1, "$**" : 2}})
```

`"whatever"` 可以指代任何东西。在设置权重时指明了是对所有字段进行索引，因此 MongoDB 并不要求你明确给出字段列表。

6.3.1　搜索语法

默认情况下，MongoDB 会使用 OR 连接查询中的每个词：“ask OR hn”。这是执行全文本查询最有效的方式，但是也可以进行短语的精确匹配，以及使用 NOT。为了精确查询“ask hn”这个短语，可以用双引号将查询内容括起来：

```
> db.runCommand({text: "hn", search: "\"ask hn\""})
{
    "queryDebugString" : "ask|hn||||ask hn||",
    "language" : "english",
    "results" : [
        {
            "score" : 2.25,
            "obj" : {
                "_id" : ObjectId("50dcab296803fa7e4f000011"),
```

```
                "title" : "Ask HN: Most valuable skills you have?",
                "url" : "/comments/4974230",
                "id" : 4974230,
                "commentCount" : 37,
                "points" : 31,
                "postedAgo" : "2 hours ago",
                "postedBy" : "bavidar"
            }
        }
    ],
    "stats" : {
        "nscanned" : 4,
        "nscannedObjects" : 0,
        "n" : 1,
        "nfound" : 1,
        "timeMicros" : 20392
    },
    "ok" : 1
}
```

这比使用 OR 的匹配慢一些，因为 MongoDB 首先要执行一个 OR 匹配，然后再对匹配结果进行 AND 匹配。

可以将查询字符串的一部分指定为字面量匹配，另一部分仍然是普通匹配：

```
> db.runCommand({text: "hn", search: "\"ask hn\" ipod"})
```

这会精确搜索 `"ask hn"` 这个短语，也会可选地搜索 `"ipod"`。

也可以使用 `"-"` 字符指定特定的词*不要*出现在搜索结果中：

```
> db.runCommand({text: "hn", search: "-startup vc"})
```

这样就会返回匹配“vc”但是不包含“startup”这个词的文档。

6.3.2 优化全文本搜索

有几种方式可以优化全文本搜索。如果能够使用某些查询条件将搜索结果的范围变小，可以创建一个由其他查询条件前缀和全文本字段组成的复合索引：

```
> db.blog.ensureIndex({"date" : 1, "post" : "text"})
```

这就是*局部的*全文本索引，MongoDB 会基于上面例子中的 `"date"` 先将搜索范围分散为多个比较小的树。这样，对于特定日期的文档进行全文本查询就会快很多了。

也可以使用其他查询条件后缀，使索引能够覆盖查询。例如，如果要返回 `"author"` 和 `"post"` 字段，可以基于这两个字段创建一个复合索引：

```
> db.blog.ensureIndex({"post" : "text", "author" : 1})
```

前缀和后缀形式也可以组合在一起使用：

```
> db.blog.ensureIndex({"date" : 1, "post" : "text", "author" : 1})
```

这里的前缀索引字段和后缀索引字段都不可以是多键字段。

创建全文本索引会自动在集合上启用 usePowerOf2Sizes 选项，这个选项可以控制空间的分配方式。这个选项能够提高写入速度，所以不要禁用它。

6.3.3 在其他语言中搜索

当一个文档被插入之后（或者索引第一次被创建之后），MongoDB 会查找索引字段，对字符串进行分词，将其减小为一个基本单元（essential unit）。然后，不同语言的分词机制是不同的，所以必须指定索引或者文档使用的语言。文本类型的索引允许指定 "default_language" 选项，它的默认值是 "english"，可以被设置为多种其他语言（MongoDB 的在线文档提供了最新的支持语言列表）。

例如，要创建一个法语的索引，可以这么做：

```
> db.users.ensureIndex({"profil" : "text", "intérêts" : "text"},
... {"default_language" : "french"})
```

这样，这个索引就会默认使用法语的分词机制，除非指定了其他的分词机制。如果在插入文档时指定 "language" 字段，就可以为每个文档分别指定分词时使用的语言：

```
> db.users.insert({"username" : "swedishChef",
... "profile" : "Bork de bork", language : "swedish"})
```

6.4 地理空间索引

MongoDB 支持几种类型的地理空间索引。其中最常用的是 2dsphere 索引（用于地球表面类型的地图）和 2d 索引（用于平面地图和时间连续的数据）。

2dsphere 允许使用 GeoJSON 格式（http://www.geojson.org）指定点、线和多边形。点可以用形如 [longitude, latitude]（[经度，纬度]）的两个元素的数组表示：

```
{
    "name" : "New York City",
    "loc" : {
        "type" : "Point",
        "coordinates" : [50, 2]
```

```
    }
}
```

线可以用一个由点组成的数组来表示：

```
{
    "name" : "Hudson River",
    "loc" : {
        "type" : "Line",
        "coordinates" : [[0,1], [0,2], [1,2]]
    }
}
```

多边形的表示方式与线一样（都是一个由点组成的数组），但是 "type" 不同：

```
{
    "name" : "New England",
    "loc" : {
        "type" : "Polygon",
        "coordinates" : [[0,1], [0,2], [1,2]]
    }
}
```

"loc" 字段的名字可以是任意的，但是其中的子对象是由 GeoJSON 指定的，不能改变。

在 ensureIndex 中使用 "2dsphere" 选项就可以创建一个地理空间索引：

```
> db.world.ensureIndex({"loc" : "2dsphere"})
```

6.4.1 地理空间查询的类型

可以使用多种不同类型的地理空间查询：交集（intersection）、包含（within）以及接近（nearness）。查询时，需要将希望查找的内容指定为形如 {"$geometry" : geoJsonDesc} 的 GeoJSON 对象。

例如，可以使用 "$geoIntersects" 操作符找出与查询位置相交的文档：

```
> var eastVillage = {
... "type" : "Polygon",
... "coordinates" : [
...     [-73.9917900, 40.7264100],
...     [-73.9917900, 40.7321400],
...     [-73.9829300, 40.7321400],
...     [-73.9829300, 40.7264100]
... ]}
> db.open.street.map.find(
... {"loc" : {"$geoIntersects" : {"$geometry" : eastVillage}}})
```

这样就会找到所有与 East Village 区域有交集的文档。

可以使用 "$within" 查询完全包含在某个区域的文档，例如："East Village 有哪些餐馆？"

```
> db.open.street.map.Ǹnd({"loc" : {"$within" : {"$geometry" : eastVillage}}})
```

与第一个查询不同，这次不会返回那些只是经过 East Village（比如街道）或者部分重叠（比如用于表示曼哈顿的多边形）的文档。

最后，可以使用 "$near" 查询附近的位置：

```
> db.open.street.map.Ǹnd({"loc" : {"$near" : {"$geometry" : eastVillage}}})
```

注意，"$near" 是唯一一个会对查询结果进行自动排序的地理空间操作符："$near" 的返回结果是按照距离由近及远排序的。

地理位置查询有一点非常有趣：不需要地理空间索引就可以使用 "$geoIntersects" 或者 "$within"（"$near" 需要使用索引）。但是，建议在用于表示地理位置的字段上建立地理空间索引，这样可以显著提高查询速度。

6.4.2　复合地理空间索引

如果有其他类型的索引，可以将地理空间索引与其他字段组合在一起使用，以便对更复杂的查询进行优化。上面提到过一种可能的查询："East Village 有哪些餐馆？"。如果仅仅使用地理空间索引，我们只能查找到 East Village 内的所有东西，但是如果要将"restaurants"或者是"pizza"单独查询出来，就需要使用其他索引中的字段了：

```
> db.open.street.map.ensureIndex({"tags" : 1, "location" : "2dsphere"})
```

然后就能够很快地找到 East Village 内的披萨店了：

```
> db.open.street.map.Ǹnd({"loc" : {"$within" : {"$geometry" : eastVillage}},
... "tags" : "pizza"})
```

其他索引字段可以放在 "2dsphere" 字段前面也可以放在后面，这取决于我们希望首先使用其他索引的字段进行过滤还是首先使用位置进行过滤。应该将那个能够过滤掉尽可能多的结果的字段放在前面。

6.4.3　2d索引

对于非球面地图（游戏地图、时间连续的数据等），可以使用 "2d" 索引代替 "2dsphere"：

```
> db.hyrule.ensureIndex({"tile" : "2d"})
```

"2d" 索引用于扁平表面，而不是球体表面。"2d" 索引不应该用在球体表面上，否则极点附近会出现大量的扭曲变形。

文档中应该使用包含两个元素的数组表示 2d 索引字段（写作本书时，这个字段还不是 GeoJSON 文档）。示例如下：

```
{
    "name" : "Water Temple",
    "tile" : [ 32, 22 ]
}
```

"2d" 索引只能对点进行索引。可以保存一个由点组成的数组，但是它只会被保存为由点组成的数组，不会被当成线。特别是对于 "$within" 查询来说，这是一项重要的区别。如果将街道保存为由点组成的数组，那么如果其中的某个点位于给定的形状之内，这个文档就会与 $within 相匹配。但是，由这些点组成的线并不一定完全包含在这个形状之内。

默认情况下，地理空间索引是假设你的值都介于 –180 ～ 180。可以根据需要在 ensureIndex 中设置更大或者更小的索引边界值：

```
> db.star.trek.ensureIndex({"light-years" : "2d"}, {"min" : -1000, "max" : 1000})
```

这会创建一个 2000 × 2000 大小的空间索引。

使用 "2d" 索引进行查询比使用 "2dsphere" 要简单许多。可以直接使用 "$near" 或者 "$within"，而不必带有 "$geometry" 子对象。可以直接指定坐标：

```
> db.hyrule.find({"tile" : {"$near" : [20, 21]}})
```

这样会返回 hyrule 集合内的全部文档，按照距离 (20,21) 这个点的距离排序。如果没有指定文档数量限制，默认最多返回 100 个文档。如果不需要这么多结果，应该根据需要设置返回文档的数量以节省服务器资源。例如，下面的代码只会返回距离 (20,21) 最近的 10 个文档：

```
> db.hyrule.find({"tile" : {"$near" : [20, 21]}}).limit(10)
```

"$within" 可以查询出某个形状（矩形、圆形或者是多边形）范围内的所有文档。如果要使用矩形，可以指定 "$box" 选项：

```
> db.hyrule.Ǹnd({"tile" : {"$within" : {"$box" : [[10, 20], [15, 30]]}}})
```

"$box" 接受一个两元素的数组：第一个元素指定左下角的坐标，第二个元素指定

右上角的坐标。

类似地，可以使用 "$center" 选项返回圆形范围内的所有文档，这个选项也是接受一个两元素数组作为参数：第一个元素是一个点，用于指定圆心；第二个参数用于指定半径：

```
> db.hyrule.find({"tile" : {"$within" : {"$center" : [[12, 25], 5]}}})
```

还可以使用多个点组成的数组来指定多边形：

```
> db.hyrule.find(
... {"tile" : {"$within" : {"$polygon" : [[0, 20], [10, 0], [-10, 0]]}}})
```

这个例子会查询出包含给定三角形内的点的所有文档。列表中的最后一个点会被连接到第一个点，以便组成多边形。

6.5 使用GridFS存储文件

GridFS 是 MongoDB 的一种存储机制，用来存储大型二进制文件。下面列出了使用 GridFS 作为文件存储的理由。

- 使用 GridFS 能够简化你的栈。如果已经在使用 MongoDB，那么可以使用 GridFS 来代替独立的文件存储工具。
- GridFS 会自动平衡已有的复制或者为 MongoDB 设置的自动分片，所以对文件存储做故障转移或者横向扩展会更容易。
- 当用于存储用户上传的文件时，GridFS 可以比较从容地解决其他一些文件系统可能会遇到的问题。例如，在 GridFS 文件系统中，如果在同一个目录下存储大量的文件，没有任何问题。
- 在 GridFS 中，文件存储的集中度会比较高，因为 MongoDB 是以 2 GB 为单位来分配数据文件的。

GridFS 也有一些缺点。

- GridFS 的性能比较低：从 MongoDB 中访问文件，不如直接从文件系统中访问文件速度快。
- 如果要修改 GridFS 上的文档，只能先将已有文档删除，然后再将整个文档重新保存。MongoDB 将文件作为多个文档进行存储，所以它无法在同一时间对文件中的所有块加锁。

通常来说，如果你有一些不常改变但是经常需要连续访问的大文件，那么使用 GridFS 再合适不过了。

6.5.1 GridFS入门

使用 GridFS 最简单的方式是使用 `mongofiles` 工具。所有的 MongoDB 发行版中都包含了 `mongofiles`，可以用它在 GridFS 中上传文件、下载文件、查看文件列表、搜索文件，以及删除文件。

与其他的命令行工具一样，运行 `mongofiles --help` 就可以查看它的可用选项了。

在下面这个会话中，首先用 `mongofiles` 从文件系统中上传一个文件到 GridFS，然后列出 GridFS 中的所有文件，最后再将之前上传过的文件从 GridFS 中下载下来：

```
$ echo "Hello, world" > foo.txt
$ ./mongofiles put foo.txt
connected to: 127.0.0.1
added file: { _id: ObjectId('4c0d2a6c3052c25545139b88'),
              filename: "foo.txt", length: 13, chunkSize: 262144,
              uploadDate: new Date(1275931244818),
              md5: "a7966bf58e23583c9a5a4059383ff850" }
done!
$ ./mongofiles list
connected to: 127.0.0.1
foo.txt 13
$ rm foo.txt
$ ./mongofiles get foo.txt
connected to: 127.0.0.1
done write to: foo.txt
$ cat foo.txt
Hello, world
```

在上面的例子中，使用 `mongofiles` 执行了三种基本操作：`put`、`list` 和 `get`。`put` 操作可以将文件系统中选定的文件上传到 GridFS；`list` 操作可以列出 GridFS 中的文件；`get` 操作与 `put` 相反，用于将 GridFS 中的文件下载到文件系统中。`mongofiles` 还支持另外两种操作：用于在 GridFS 中搜索文件的 `search` 操作和用于从 GridFS 中删除文件的 `delete` 操作。

6.5.2 在MongoDB驱动程序中使用GridFS

所有客户端驱动程序都提供了 GridFS API。例如，可以用 PyMongo（MongoDB 的 Python 驱动程序）执行与上面直接使用 `mongofiles` 一样的操作：

```
>>> from pymongo import Connection
>>> import gridfs
>>> db = Connection().test
>>> fs = gridfs.GridFS(db)
>>> file_id = fs.put("Hello, world", filename="foo.txt")
>>> fs.list()
[u'foo.txt']
>>> fs.get(file_id).read()
```

```
'Hello, world'
```

PyMongo 中用于操作 GridFS 的 API 与 `mongofiles` 非常像：可以很方便地执行 `put`、`get` 和 `list` 操作。几乎所有 MongoDB 驱动程序都遵循这种基本模式对 GridFS 进行操作，当然通常也会提供一些更高级的功能。关于特定驱动程序对 GridFS 的操作，可以查询相关驱动程序的文件。

6.5.3 揭开GridFS的面纱

GridFS 是一种轻量级的文件存储规范，用于存储 MongoDB 中的普通文档。MongoDB 服务器几乎不会对 GridFS 请求做“特殊”处理，所有处理都由客户端的驱动程序和工具负责。

GridFS 背后的理念是：可以将大文件分割为多个比较大的块，将每个块作为独立的文档进行存储。由于 MongoDB 支持在文档中存储二进制数据，所以可以将块存储的开销降到非常低。除了将文件的每一个块单独存储之外，还有一个文档用于将这些块组织在一起并存储该文件的元信息。

GridFS 中的块会被存储到专用的集合中。块默认使用的集合是 `fs.chunks`，不过可以修改为其他集合。在块集合内部，各个文档的结构非常简单：

```
{
    "_id" : ObjectId("..."),
    "n" : 0,
    "data" : BinData("..."),
    "files_id" : ObjectId("...")
}
```

与其他的 MongoDB 文档一样，块也都拥有一个唯一的 `"_id"`。另外，还有如下几个键。

- `"files_id"`
  块所属文件的元信息。

- `"n"`
  块在文件中的相对位置。

- `"data"`
  块所包含的二进制数据。

每个文件的元信息被保存在一个单独的集合中，默认情况下这个集合是 `fs.files`。这个文件集合中的每一个文档表示 GridFS 中的一个文件，文档中可以包含与这个文

件相关的任意用户自定义元信息。除用户自定义的键之外，还有几个键是 GridFS 规范规定必须要有的。

- `"_id"`
  文件的唯一 id，这个值就是文件的每个块文档中 `"files_id"` 的值。
- `"length"`
  文件所包含的字节数。
- `"chunkSize"`
  组成文件的每个块的大小，单位是字节。这个值默认是 256 KB，可以在需要时进行调整。
- `"uploadDate"`
  文件被上传到 GridFS 的日期。
- `"md5"`
  文件内容的 md5 校验值，这个值由服务器端计算得到。

这些必须字段中最有意思（或者说能够见名知意）的一个可能是 `"md5"`。`"md5"` 字段的值是由 MongoDB 服务器使用 `filemd5` 命令得到的，这个命令可以用来计算上传到 GridFS 的块的 md5 校验值。这意味着，用户可以通过检查文件的 `md5` 校验值来确保文件上传正确。

如上面所说，在 `fs.files` 中，除了这些必须字段外，可以使用任何自定义的字段来保存必需的文件元信息。可能你希望在文件元信息中保存文件的下载次数、MIME 类型或者用户评分。

只要理解了 GridFS 底层的规范，自己就可以很容易地实现一些驱动程序没有提供的辅助功能。例如，可以使用 `distinct` 命令得到 GridFS 中保存文件的文件名集合（集合中的每个文件名都是唯一的）。

```
> db.fs.files.distinct("filename")
[ "foo.txt" , "bar.txt" , "baz.txt" ]
```

这样，在加载或者收集文件相关信息时，应用程序可以拥有非常大的灵活性。

第 7 章

聚合

如果你有数据存储在 MongoDB 中，你想做的可能就不仅仅是将数据提取出来那么简单了；你可能希望对数据进行分析并加以利用。本章介绍 MongoDB 提供的聚合工具：

- 聚合框架；
- MapReduce；
- 几个简单聚合命令：`count`、`distinct` 和 `group`。

7.1 聚合框架

使用聚合框架可以对集合中的文档进行变换和组合。基本上，可以用多个构件创建一个管道（pipeline），用于对一连串的文档进行处理。这些构件包括筛选（filtering）、投射（projecting）、分组（grouping）、排序（sorting）、限制（limiting）和跳过（skipping）。

例如，有一个保存着杂志文章的集合，你可能希望找出发表文章最多的那个作者。假设每篇文章被保存为 MongoDB 中的一个文档，可以按照如下步骤创建管道。

(1) 将每个文章文档中的作者投射出来。

(2) 将作者按照名字排序，统计每个名字出现的次数。

(3) 将作者按照名字出现次数降序排列。

(4) 将返回结果限制为前 5 个。

这里面的每一步都对应聚合框架中的一个操作符：

(1) `{"$project" : {"author" : 1}}`

这样可以将 `"author"` 从每个文档中投射出来。

这个语法与查询中的字段选择器比较像：可以通过指定 `"`*`fieldname`*`" : 1` 选择需要投射的字段，或者通过指定 `"`*`fieldname`*`" : 0` 排除不需要的字段。执行完这个 `"$project"` 操作之后，结果集中的每个文档都会以 `{"_id" : `*`id`*`, "author" : "`*`authorName`*`"}` 这样的形式表示。这些结果只会在内存中存在，不会被写入磁盘。

(2) `{"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}}`

这样就会将作者按照名字排序，某个作者的名字每出现一次，就会对这个作者的 `"count"` 加 1。

这里首先指定了需要进行分组的字段 `"author"`。这是由 `"_id" : "$author"` 指定的。可以将这个操作想象为：这个操作执行完后，每个作者只对应一个结果文档，所以 `"author"` 就成了文档的唯一标识符（`"_id"`）。

第二个字段的意思是为分组内每个文档的 `"count"` 字段加 1。注意，新加入的文档中并不会有 `"count"` 字段；这 `"$group"` 创建的一个新字段。

执行完这一步之后，结果集中的每个文档会是这样的结构：`{"_id" : "`*`authorName`*`", "count" : `*`articleCount`*`}`。

(3) `{"$sort" : {"count" : -1}}`

这个操作会对结果集中的文档根据 `"count"` 字段进行降序排列。

(4) `{"$limit" : 5}`

这个操作将最终的返回结果限制为当前结果中的前 5 个文档。

在 MongoDB 中实际运行时，要将这些操作分别传给 `aggregate()` 函数：

```
> db.articles.aggregate({"$project" : {"author" : 1}},
... {"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}},
... {"$sort" : {"count" : -1}},
... {"$limit" : 5})
{
    "result" : [
        {
            "_id" : "R. L. Stine",
```

```
            "count" : 430
        },
        {
            "_id" : "Edgar Wallace",
            "count" : 175
        },
        {
            "_id" : "Nora Roberts",
            "count" : 145
        },
        {
            "_id" : "Erle Stanley Gardner",
            "count" : 140
        },
        {
            "_id" : "Agatha Christie",
            "count" : 85
        }
    ],
    "ok" : 1
}
```

aggregate() 会返回一个文档数组，其中的内容是发表文章最多的 5 个作者。



如果管道没有给出预期的结果，就需要进行调试，调试时，可以先只指定第一个管道操作符。如果这时得到了预期结果，那就再指定第二个管道操作符。以前面的例子来说，首先要试着只使用 "$project" 操作符进行聚合；如果这个操作符的结果是有效的，就再添加 "$group" 操作符；如果结果还是有效的，就再添加 "$sort"；最后再添加 "$limit" 操作符。这样就可以逐步定位到造成问题的操作符。

本书写作时，聚合框架还不能对集合进行写入操作，因此所有结果必须返回给客户端。所以，聚合的结果必须要限制在 16 MB 以内（MongoDB 支持的最大响应消息大小）。

7.2 管道操作符

每个操作符都会接受一连串的文档，对这些文档做一些类型转换，最后将转换后的文档作为结果传递给下一个操作符（对于最后一个管道操作符，是将结果返回给客户端）。

不同的管道操作符可以按任意顺序组合在一起使用，而且可以被重复任意多次。例如，可以先做 "$match"，然后做 "$group"，然后再做 "$match"（与之前的 "$match" 匹配不同的查询条件）。

7.2.1 $match

$match 用于对文档集合进行筛选，之后就可以在筛选得到的文档子集上做聚合。例如，如果想对 Oregon（俄勒冈州，简写为 OR）的用户做统计，就可以使用 {$match : {"state" : "OR"}}。"$match" 可以使用所有常规的查询操作符（"$gt"、"$lt"、"$in" 等）。有一个例外需要注意：不能在 "$match" 中使用地理空间操作符。

通常，在实际使用中应该尽可能将 "$match" 放在管道的前面位置。这样做有两个好处：一是可以快速将不需要的文档过滤掉，以减少管道的工作量；二是如果在投射和分组之前执行 "$match"，查询可以使用索引。

7.2.2 $project

相对于“普通”的查询而言，管道中的投射操作更加强大。使用 "$project" 可以从子文档中提取字段，可以重命名字段，还可以在这些字段上进行一些有意思的操作。

最简单的一个 "$project" 操作是从文档中选择想要的字段。可以指定包含或者不包含一个字段，它的语法与查询中的第二个参数类似。如果在原来的集合上执行下面的代码，返回的结果文档中只包含一个 "author" 字段。

```
> db.articles.aggregate({"$project" : {"author" : 1, "_id" : 0}})
```

默认情况下，如果文档中存在 "_id" 字段，这个字段就会被返回（"_id" 字段可以被一些管道操作符移除，也可能已经被之前的投射操作给移除了）。可以使用上面的代码将 "_id" 从结果文档中移除。包含字段和排除字段的规则与常规查询中的语法一致。

也可以将投射过的字段进行重命名。例如，可以将每个用户文档的 "_id" 在返回结果中重命名为 "userId"：

```
> db.users.aggregate({"$project" : {"userId" : "$_id", "_id" : 0}})
{
    "result" : [
        {
            "userId" : ObjectId("50e4b32427b160e099ddbee7")
        },
        {
            "userId" : ObjectId("50e4b32527b160e099ddbee8")
        }
        ...
    ],
    "ok" : 1
}
```

这里的 "*$fieldname*" 语法是为了在聚合框架中引用 *fieldname* 字段（上面的例子中是 "_id"）的值。例如，"$age" 会被替换为 "age" 字段的内容（可能是数值，也可能是字符串），"$tags.3" 会被替换为 tags 数组中的第 4 个元素。所以，上面例子中的 "$_id" 会被替换为进入管道的每个文档的 "_id" 字段的值。

注意，必须明确指定将 "_id" 排除，否则这个字段的值会被返回两次：一次被标为 "userId"，一次被标为 "_id"。可以使用这种技术生成字段的多个副本，以便在之后的 "$group" 中使用。

在对字段进行重命名时，MongoDB 并不会记录字段的历史名称。因此，如果在 "originalfieldname" 字段上有一个索引，聚合框架无法在下面的排序操作中使用这个索引，尽管人眼一下子就能看出下面代码中的 "newfieldname" 与 "originalfieldname" 表示同一个字段。

```
> db.articles.aggregate({"$project" : {"newfieldname" : "$originalfieldname"}},
... {"$sort" : {"newfieldname" : 1}})
```

所以，应该尽量在修改字段名称之前使用索引。

1. 管道表达式

最简单的 "$project" 表达式是包含和排除字段，以及字段名称（"*$fieldname*"）。但是，还有一些更强大的选项。也可以使用表达式（expression）将多个字面量和变量组合在一个值中使用。

在聚合框架中有几个表达式可用来组合或者进行任意深度的嵌套，以便创建复杂的表达式。

2. 数学表达式（mathematical expression）

算术表达式可用于操作数值。指定一组数值，就可以使用这个表达式进行操作了。例如，下面的表达式会将 "salary" 和 "bonus" 字段的值相加。

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "totalPay" : {
...             "$add" : ["$salary", "$bonus"]
...         }
...     }
... })
```

可以将多个表达式嵌套在一起组成更复杂的表达式。假设我们想要从总金额中扣除为 401(k)[1] 缴纳的金额。可以使用 "$subtract" 表达式：

注 1：401(k) 是美国的一种养老金计划。——译者注

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "totalPay" : {
...              "$subtract" : [{"$add" : ["$salary", "$bonus"]}, "$401k"]
...         }
...     }
... })
```

表达式可以进行任意层次的嵌套。

下面是每个操作符的语法：

- `"$add" : [`*`expr1[, expr2, ..., exprN]`*`]`
  这个操作符接受一个或多个表达式作为参数，将这些表达式相加。

- `"$subtract" : [`*`expr1, expr2`*`]`
  接受两个表达式作为参数，用第一个表达式减去第二个表达式作为结果。

- `"$multiply" : [`*`expr1[, expr2, ..., exprN]`*`]`
  接受一个或者多个表达式，并且将它们相乘。

- `"$divide" : [`*`expr1, expr2`*`]`
  接受两个表达式，用第一个表达式除以第二个表达式的商作为结果。

- `"$mod" : [`*`expr1, expr2`*`]`
  接受两个表达式，将第一个表达式除以第二个表达式得到的余数作为结果。

3. 日期表达式（date expression）

许多聚合都是基于时间的：上周发生了什么？上个月发生了什么？过去一年间发生了什么？因此，聚合框架中包含了一些用于提取日期信息的表达式：`"$year"`、`"$month"`、`"$week"`、`"$dayOfMonth"`、`"$dayOfWeek"`、`"$dayOfYear"`、`"$hour"`、`"$minute"` 和 `"$second"`。只能对日期类型的字段进行日期操作，不能对数值类型字段做日期操作。

每种日期类型的操作都是类似的：接受一个日期表达式，返回一个数值。下面的代码会返回每个雇员入职的月份：

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "hiredIn" : {"$month" : "$hireDate"}
...     }
... })
```

也可以使用字面量日期。下面的代码会计算出每个雇员在公司内的工作时间：

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "tenure" : {
...             "$subtract" : [{"$year" : new Date()}, {"$year" : "$hireDate"}]
...         }
...     }
... })
```

4. 字符串表达式（string expression）

也有一些基本的字符串操作可以使用，它们的签名如下所示。

- `"$substr" : [`*`expr, startOffset, numToReturn`*`]`
  其中第一个参数 `expr` 必须是个字符串，这个操作会截取这个字符串的子串（从第 `startOffset` 字节开始的 *`numToReturn`* 字节，注意，是字节，不是字符。在多字节编码中尤其要注意这一点）`expr` 必须是字符串。

- `"$concat" : [`*`expr1[, expr2, ..., exprN]`*`]`
  将给定的表达式（或者字符串）连接在一起作为返回结果。

- `"$toLower" :` *`expr`*
  参数 `expr` 必须是个字符串值，这个操作返回 `expr` 的小写形式。

- `"$toUpper" :` *`expr`*
  参数 `expr` 必须是个字符串值，这个操作返回 `expr` 的大写形式。

改变字符大小写的操作，只保证对罗马字符有效。

下面是一个生成 j.doe@example.com 格式的 email 地址的例子。它提取 `"$firstname"` 的第一个字符，将其与多个常量字符串和 `"$lastname"` 连接成一个字符串：

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "email" : {
...             "$concat" : [
...                 {"$substr" : ["$firstName", 0, 1]},
...                 ".",
...                 "$lastName",
...                 "@example.com"
...             ]
...         }
...     }
... })
```

5. 逻辑表达式（logical expression）

有一些逻辑表达式可以用于控制语句。

下面是几个比较表达式。

- `"$cmp" : [`*`expr1, expr2`*`]`
  比较 `expr1` 和 `expr2`。如果 `expr1` 等于 `expr2`，返回 0；如果 `expr1` < `expr2`，返回一个负数；如果 `expr1` >`expr2`，返回一个正数。

- `"$strcasecmp" : [`*`string1, string2`*`]`
  比较 `string1` 和 `string2`，区分大小写。只对罗马字符组成的字符串有效。

- `"$eq"/"$ne"/"$gt"/"$gte"/"$lt"/"$lte" : [`*`expr1, expr2`*`]`
  对 `expr1` 和 `expr2` 执行相应的比较操作，返回比较的结果（`true` 或 `false`）。

下面是几个布尔表达式。

- `"$and" : [`*`expr1[, expr2, ..., exprN]`*`]`
  如果所有表达式的值都是 `true`，那就返回 `true`，否则返回 `false`。

- `"$or" : [`*`expr1[, expr2, ..., exprN]`*`]`
  只要有任意表达式的值为 `true`，就返回 `true`，否则返回 `false`。

- `"$not" :` *`expr`*
  对 `expr` 取反。

还有两个控制语句。

- `"$cond" : [`*`booleanExpr, trueExpr, falseExpr`*`]`
  如果 `booleanExpr` 的值是 `true`，那就返回 `trueExpr`，否则返回 `falseExpr`。

- `"$ifNull" : [`*`expr, replacementExpr`*`]`
  如果 `expr` 是 `null`，返回 `replacementExpr`，否则返回 `expr`。

通过这些操作符，就可以在聚合中使用更复杂的逻辑，可以对不同数据执行不同的代码，得到不同的结果。

管道对于输入数据的形式有特定要求，所以这些操作符在传入数据时要特别注意。算术操作符必须接受数值，日期操作符必须接受日期，字符串操作符必须接受字符串，如果有字符缺失，这些操作符就会报错。如果你的数据集不一致，可以通过这个条件来检测缺失的值，并且进行填充。

6. 一个提取的例子

假如有个教授想通过某种比较复杂的计算为学生打分：出勤率占 10%，日常测验成绩占 30%，期末考试占 60%（如果是老师最宠爱的学生，那么分数就是 100）。可以使用如下代码：

```
> db.students.aggregate(
... {
...     "$project" : {
...         "grade" : {
...             "$cond" : [
...                 "$teachersPet",
...                 100, // if
...                 {    // else
...                     "$add" : [
...                         {"$multiply" : [.1, "$attendanceAvg"]},
...                         {"$multiply" : [.3, "$quizzAvg"]},
...                         {"$multiply" : [.6, "$testAvg"]}
...                     ]
...                 }
...             ]
...         }
...     }
... })
```

7.2.3 $group

`$group` 操作可以将文档依据特定字段的不同值进行分组。下面是几个分组的例子。

- 如果我们以分钟作为计量单位，希望找出每天的平均湿度，就可以根据 `"day"` 字段进行分组。
- 如果有一个学生集合，希望按照分数等级将学生分为多个组，可以根据 `"grade"` 字段进行分组。
- 如果有一个用户集合，希望知道每个城市有多少用户，可以根据 `"state"` 和 `"city"` 两个字段对集合进行分组，每个 `"city"`/`"state"` 对对应一个分组。不应该只根据 `"city"` 字段进行分组，因为不同的州可能拥有相同名字的城市。

如果选定了需要进行分组的字段，就可以将选定的字段传递给 `"$group"` 函数的 `"_id"` 字段。对于上面的例子，相应的代码如下：

- `{"$group" : {"_id" : "$day"}}`
- `{"$group" : {"_id" : "$grade"}}`
- `{"$group" : {"_id" : {"state" : "$state", "city" : "$city"}}}`

如果执行这些代码，结果集中每个分组对应一个只有一个字段（分组键）的文档。例如，按学生分数等级进行分组的结果可能是：`{"result" : [{"_id" :`

"A+"}, {"_id" : "A"}, {"_id" : "A-"}, ..., {"_id" : "F"}], "ok" : 1}。通过上面这些代码，可以得到特定字段中每一个不同的值，但是所有例子都要求基于这些分组进行一些计算。因此，可以添加一些字段，使用分组操作符对每个分组中的文档做一些计算。

1. 分组操作符

这些分组操作符允许对每个分组进行计算，得到相应的结果。7.1 节介绍过 "$sum" 分组操作符的作用：分组中每出现一个文档，它就对计算结果加 1，这样便可以得到每个分组中的文档数量。

2. 算术操作符

有两个操作符可以用于对数值类型字段的值进行计算："$sum" 和 "$average"。

- "$sum" : *value*

  对于分组中的每一个文档，将 value 与计算结果相加。注意，上面的例子中使用了一个字面量数字 1，但是这里也可以使用比较复杂的值。例如，如果有一个集合，其中的内容是各个国家的销售数据，使用下面的代码就可以得到每个国家的总收入：

  ```
  > db.sales.aggregate(
  ... {
  ...     "$group" : {
  ...         "_id" : "$country",
  ...         "totalRevenue" : {"$sum" : "$revenue"}
  ...     }
  ... })
  ```

- "$avg" : *value*

  返回每个分组的平均值。

  例如，下面的代码会返回每个国家的平均收入，以及每个国家的销量：

  ```
  > db.sales.aggregate(
  ... {
  ...     "$group" : {
  ...         "_id" : "$country",
  ...         "totalRevenue" : {"$avg" : "$revenue"},
  ...         "numSales" : {"$sum" : 1}
  ...     }
  ... })
  ```

3. 极值操作符（extreme operator）

下面的四个操作符可用于得到数据集合中的“边缘”值。

- `"$max" : expr`
  返回分组内的最大值。
- `"$min" : expr`
  返回分组内的最小值。
- `"$first" : expr`
  返回分组的第一个值，忽略后面所有值。只有排序之后，明确知道数据顺序时这个操作才有意义。
- `"$last" : expr`
  与 `"$first"` 相反，返回分组的最后一个值。

`"$max"` 和 `"$min"` 会查看每一个文档，以便得到极值。因此，如果数据是无序的，这两个操作符也可以有效工作；如果数据是有序的，这两个操作符就会有些浪费。假设有一个存有学生考试成绩的数据集，需要找到其中的最高分与最低分：

```
> db.scores.aggregate(
... {
...     "$group" : {
...         "_id" : "$grade",
...         "lowestScore" : {"$min" : "$score"},
...         "highestScore" : {"$max" : "$score"}
...     }
... })
```

另一方面，如果数据集是按照希望的字段排序过的，那么 `"$first"` 和 `"$last"` 操作符就会非常有用。下面的代码与上面的代码可以得到同样的结果：

```
> db.scores.aggregate(
... {
...     "$sort" : {"score" : 1}
... },
... {
...     "$group" : {
...         "_id" : "$grade",
...         "lowestScore" : {"$first" : "$score"},
...         "highestScore" : {"$last" : "$score"}
...     }
... })
```

如果数据是排过序的，那么 `$first` 和 `$last` 会比 `$min` 和 `$max` 效率更高。如果不准备对数据进行排序，那么直接使用 `$min` 和 `$max` 会比先排序再使用 `$first` 和 `$last` 效率更高。

4. 数组操作符

有两个操作符可以进行数组操作。

- `"$addToSet" : expr`

  如果当前数组中不包含 `expr`，那就将它添加到数组中。在返回结果集中，每个元素最多只出现一次，而且元素的顺序是不确定的。

- `"$push" : expr`

  不管 `expr` 是什么值，都将它添加到数组中。返回包含所有值的数组。

5. 分组行为

有两个操作符不能用前面介绍的流式工作方式对文档进行处理，`"$group"` 是其中之一。大部分操作符的工作方式都是流式的，只要有新文档进入，就可以对新文档进行处理，但是 `"$group"` 必须要等收到所有的文档之后，才能对文档进行分组，然后才能将各个分组发送给管道中的下一个操作符。这意味着，在分片的情况下，`"$group"` 会先在每个分片上执行，然后各个分片上的分组结果会被发送到 mongos 再进行最后的统一分组，剩余的管道工作也都是在 mongos（而不是在分片）上运行的。

7.2.4 $unwind

拆分（unwind）可以将数组中的每一个值拆分为单独的文档。例如，如果有一篇拥有多条评论的博客文章，可以使用 `$unwind` 将每条评论拆分为一个独立的文档：

```
> db.blog.findOne()
{
    "_id" : ObjectId("50eeffc4c82a5271290530be"),
    "author" : "k",
    "post" : "Hello, world!",
    "comments" : [
        {
            "author" : "mark",
            "date" : ISODate("2013-01-10T17:52:04.148Z"),
            "text" : "Nice post"
        },
        {
            "author" : "bill",
            "date" : ISODate("2013-01-10T17:52:04.148Z"),
            "text" : "I agree"
        }
    ]
}
> db.blog.aggregate({"$unwind" : "$comments"})
{
    "results" :
        {
            "_id" : ObjectId("50eeffc4c82a5271290530be"),
```

```
            "author" : "k",
            "post" : "Hello, world!",
            "comments" : {
                "author" : "mark",
                "date" : ISODate("2013-01-10T17:52:04.148Z"),
                "text" : "Nice post"
            }
        },
        {
            "_id" : ObjectId("50eeffc4c82a5271290530be"),
            "author" : "k",
            "post" : "Hello, world!",
            "comments" : {
                "author" : "bill",
                "date" : ISODate("2013-01-10T17:52:04.148Z"),
                "text" : "I agree"
            }
        }
    ],
    "ok" : 1
}
```

如果希望在查询中得到特定的子文档，这个操作符就会非常有用：先使用 "$unwind" 得到所有子文档，再使用 "$match" 得到想要的文档。例如，如果要得到特定用户的所有评论（只需要得到评论，不需要返回评论所属的文章），使用普通的查询是不可能做到的。但是，通过提取、拆分、匹配，就很容易了：

```
> db.blog.aggregate({"$project" : {"comments" : "$comments"}},
... {"$unwind" : "$comments"},
... {"$match" : {"comments.author" : "Mark"}})
```

由于最后得到的结果仍然是一个 "comments" 子文档，所以你可能希望再做一次投射，以便让输出结果更优雅。

7.2.5 $sort

可以根据任何字段（或者多个字段）进行排序，与在普通查询中的语法相同。如果要对大量的文档进行排序，强烈建议在管道的第一阶段进行排序，这时的排序操作可以使用索引。否则，排序过程就会比较慢，而且会占用大量内存。

可以在排序中使用文档中实际存在的字段，也可以使用在投射时重命名的字段：

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "compensation" : {
...             "$add" : ["$salary", "$bonus"]
...         },
...         "name" : 1
```

```
...     }
... },
... {
...     "$sort" : {"compensation" : -1, "name" : 1}
... })
```

这个例子会对员工排序，最终的结果是按照报酬从高到低，姓名从 A 到 Z 的顺序排列。

排序方向可以是 1（升序）和 −1（降序）。

与前面讲过的 `"$group"` 一样，`"$sort"` 也是一个无法使用流式工作方式的操作符。`"$sort"` 也必须要接收到所有文档之后才能进行排序。在分片环境下，先在各个分片上进行排序，然后将各个分片的排序结果发送到 mongos 做进一步处理。

7.2.6 `$limit`

`$limit` 会接受一个数字 *n*，返回结果集中的前 *n* 个文档。

7.2.7 `$skip`

`$skip` 也是接受一个数字 *n*，丢弃结果集中的前 *n* 个文档，将剩余文档作为结果返回。在“普通”查询中，如果需要跳过大量的数据，那么这个操作符的效率会很低。在聚合中也是如此，因为它必须要先匹配到所有需要跳过的文档，然后再将这些文档丢弃。

7.2.8 使用管道

应该尽量在管道的开始阶段（执行 `"$project"`、`"$group"` 或者 `"$unwind"` 操作之前）就将尽可能多的文档和字段过滤掉。管道如果不是直接从原先的集合中使用数据，那就无法在筛选和排序中使用索引。如果可能，聚合管道会尝试对操作进行排序，以便能够有效使用索引。

MongoDB 不允许单一的聚合操作占用过多的系统内存：如果 MongoDB 发现某个聚合操作占用了 20% 以上的内存，这个操作就会直接输出错误。允许将输出结果利用管道放入一个集合中是为了方便以后使用（这样可以将所需的内存减至最小）。

如果能够通过 `"$match"` 操作迅速减小结果集的大小，就可以使用管道进行实时聚合。由于管道会不断包含更多的文档，会越来越复杂，所以几乎不可能实时得到管道的操作结果。

7.3 MapReduce

MapReduce 是聚合工具中的明星，它非常强大、非常灵活。有些问题过于复杂，无法使用聚合框架的查询语言来表达，这时可以使用 MapReduce。MapReduce 使用 JavaScript 作为“查询语言”，因此它能够表达任意复杂的逻辑。然而，这种强大是有代价的：MapReduce 非常慢，不应该用在实时的数据分析中。

MapReduce 能够在多台服务器之间并行执行。它会将一个大问题拆分为多个小问题，将各个小问题发送到不同的机器上，每台机器只负责完成一部分工作。所有机器都完成时，再将这些零碎的解决方案合并为一个完整的解决方案。

MapReduce 需要几个步骤。最开始是映射（map），将操作映射到集合中的每个文档。这个操作要么“无作为”，要么“产生一些键和 *X* 个值”。然后就是中间环节，称作洗牌（shuffle），按照键分组，并将产生的键值组成列表放到对应的键中。化简（reduce）则把列表中的值化简成一个单值。这个值被返回，然后接着进行洗牌，直到每个键的列表只有一个值为止，这个值也就是最终结果。

下面会多举几个 MapReduce 的例子，这个工具非常强大，但也有点复杂。

7.3.1 示例1：找出集合中的所有键

用 MapReduce 来解决这个问题有点大材小用，不过还是一种了解其机制的不错的方式。要是已经知道 MapReduce 的原理，则直接跳到本节最后，看看 MongoDB 中 MapReduce 的使用注意事项。

MongoDB 会假设你的模式是动态的，所以并不跟踪记录每个文档中的键。通常找到集合中所有文档所有键的最好方式就是用 MapReduce。在本例中，会记录每个键出现了多少次。内嵌文档中的键就不计算了，但给 `map` 函数做个简单修改就能实现这个功能了。

在映射环节，我们希望得到集合中每个文档的所有键。`map` 函数使用特别的 `emit` 函数“返回”要处理的值。`emit` 会给 MapReduce 一个键（类似于前面 `$group` 所使用的键）和一个值。这里用 `emit` 将文档某个键的计数（count）返回（`{count : 1}`）。我们想为每个键单独计数，所以为文档中的每个键调用一次 `emit`。`this` 就是当前映射文档的引用：

```
> map = function() {
... for (var key in this) {
...     emit(key, {count : 1});
... }};
```

这样就有了许许多多 {count : 1} 文档，每一个都与集合中的一个键相关。这种由一个或多个 {count : 1} 文档组成的数组，会传递给 reduce 函数。reduce 函数有两个参数，一个是 key，也就是 emit 返回的第一个值，还有另外一个数组，由一个或者多个与键对应的 {count : 1} 文档组成。

```
> reduce = function(key, emits) {
... total = 0;
... for (var i in emits) {
...     total += emits[i].count;
... }
... return {"count" : total};
... }
```

reduce 一定要能够在之前的 map 阶段或者前一个 reduce 阶段的结果上反复执行。所以 reduce 返回的文档必须能作为 reduce 的第二个参数的一个元素。例如，x 键映射到了 3 个文档 {count : 1，id : 1}、{count : 1，id : 2} 和 {count : 1，id : 3}，其中 id 键只用于区分不同的文档。MongoDB 可能会这样调用 reduce：

```
> r1 = reduce("x", [{count : 1, id : 1}, {count : 1, id : 2}])
{count : 2}
> r2 = reduce("x", [{count : 1, id : 3}])
{count : 1}
> reduce("x", [r1, r2])
{count : 3}
```

不能认为第二个参数总是初始文档之一（比如 {count : 1}）或者长度固定。reduce 应该能处理 emit 文档和其他 reduce 返回结果的各种组合。

总之，MapReduce 函数可能会是下面这样：

```
> mr = db.runCommand({"mapreduce" : "foo", "map" : map, "reduce" : reduce})
{
    "result" : "tmp.mr.mapreduce_1266787811_1",
    "timeMillis" : 12,
    "counts" : {
        "input" : 6
        "emit" : 14
        "output" : 5
    },
    "ok" : true
}
```

MapReduce 返回的文档包含很多与操作有关的元信息。

- "result" : "tmp.mr.mapreduce_1266787811_1"
  这是存放 MapReduce 结果的集合名。这是个临时集合，MapReduce 的连接关闭后它就被自动删除了。本章稍后会介绍如何指定一个好一点的名字以及将结果集合持久化。

- `"timeMillis" : 12`

  操作花费的时间，单位是毫秒。

- `"counts" : { ... }`

  这个内嵌文档主要用作调试，其中包含 3 个键。

  - `"input" : 6`

    发送到 `map` 函数的文档个数。

  - `"emit" : 14`

    在 `map` 函数中 `emit` 被调用的次数。

  - `"output" : 5`

    结果集合中的文档数量。

对结果集合进行查询会发现原有集合的所有键及其计数：

```
> db[mr.result].find()
{ "_id" : "_id", "value" : { "count" : 6 } }
{ "_id" : "a", "value" : { "count" : 4 } }
{ "_id" : "b", "value" : { "count" : 2 } }
{ "_id" : "x", "value" : { "count" : 1 } }
{ "_id" : "y", "value" : { "count" : 1 } }
```

这个结果集中的每个 `"_id"` 对应原集合中的一个键，`"value"` 键的值就是 `reduce` 的最终结果。

7.3.2 示例2：网页分类

假设有个网站，人们可以提交其他网页的链接，比如 reddit（http://www.reddit.com）。提交者可以给这个链接添加标签，表明主题，比如 politics、geek 或者 icanhascheezburger。可以用 MapReduce 找出哪个主题最为热门，热门与否由最近的投票决定。

首先，建立一个 `map` 函数，发出（emit）标签和一个基于流行度和新旧程度的值。

```
map = function() {
    for (var i in this.tags) {
        var recency = 1/(new Date() - this.date);
        var score = recency * this.score;

        emit(this.tags[i], {"urls" : [this.url], "score" : score});
    }
};
```

现在就化简同一个标签的所有值，以得到这个标签的分数：

```
reduce = function(key, emits) {
    var total = {urls : [], score : 0}
    for (var i in emits) {
        emits[i].urls.forEach(function(url) {
            total.urls.push(url);
        }
        total.score += emits[i].score;
    }
    return total;
};
```

最终的集合包含每个标签的 URL 列表和表示该标签流行程度的分数。

7.3.3 MongoDB和MapReduce

前面两个例子只用到了 `mapreduce`、`map` 和 `reduce` 键。这 3 个键是必需的，但是 MapReduce 命令还有很多可选的键。

- `"finalize" : ` *`function`*
  可以将 `reduce` 的结果发送给这个键，这是整个处理过程的最后一步。

- `"keeptemp" : ` *`boolean`*
  如果为值为 `true`，那么在连接关闭时会将临时结果集合保存下来，否则不保存。

- `"out" : ` *`string`*
  输出集合的名称。如果设置了这选项，系统会自动设置 `keeptemp : true`。

- `"query" : ` *`document`*
  在发往 `map` 函数前，先用指定条件过滤文档。

- `"sort" : ` *`document`*
  在发往 `map` 前先给文档排序（与 `limit` 一同使用非常有用）。

- `"limit" : ` *`integer`*
  发往 `map` 函数的文档数量的上限。

- `"scope" : ` *`document`*
  可以在 JavaScript 代码中使用的变量。

- `"verbose" : ` *`boolean`*
  是否记录详细的服务器日志。

1. finalize函数

和 group 命令一样，MapReduce 也可以使用 finalize 函数作为参数。它会在最后一个 reduce 输出结果后执行，然后将结果存到临时集合中。

返回体积比较大的结果集对 MapReduce 不是什么大不了的事情，因为它不像 group 那样有 4 MB 的限制。然而，信息总是要传递出去的，通常来说，finalize 是计算平均数、裁剪数组、清除多余信息的好时机。

2. 保存结果集合

默认情况下，Mongo 会在执行 MapReduce 时创建一个临时集合，集合名是系统选的一个不太常用的名字，将 "mr"、执行 MapReduce 的集合名、时间戳以及数据库作业 ID，用“.”连成一个字符串，这就是临时集合的名字。结果产生形如 mr.stuff.18234210220.2 这样的名字。MongoDB 会在调用的连接关闭时自动销毁这个集合（也可以在用完之后手动删除）。如果希望保存这个集合，就要将 keeptemp 选项指定为 true。

如果要经常使用这个临时集合，你可能想给它起个好点的名字。利用 out 选项（该选项接受字符串作为参数）就可以为临时集合指定一个易读易懂的名字。如果用了 out 选项，就不必指定 keeptemp : true 了，因为指定 out 选项时系统会将 keeptemp 设置为 true。即便你取了一个非常好的名字，MongoDB 也会在 MapReduce 的中间过程使用自动生成的集合名。处理完成后，会自动将临时集合的名字更改为你指定的集合名，这个重命名的过程是原子性的。也就是说，如果多次对同一个集合调用 MapReduce，也不会在操作中遇到集合不完整的情况。

MapReduce 产生的集合就是一个普通的集合，在这个集合上执行 MapReduce 完全没有问题，或者在前一个 MapReduce 的结果上执行 MapReduce 也没有问题，如此往复直到无穷都没问题！

3. 对文档子集执行MapReduce

有时需要对集合的一部分执行 MapReduce。只需在传给 map 函数前使用查询对文档进行过滤就好了。

每个传递给 map 函数的文档都要先反序列化，从 BSON 对象转换为 JavaScript 对象，这个过程非常耗时。如果事先知道只需要对集合的一部分文档执行 MapReduce，那么在 map 之前先对文档进行过滤可以极大地提高 map 速度。可以通过 "query"、"limit" 和 "sort" 等键对文档进行过滤。

"query" 键的值是一个查询文档。通常查询返回的结果会传递给 map 函数。例如，有一个做跟踪分析的应用程序，现在我们需要上周的总结摘要，只要使用如下命令

对上周的文档执行 MapReduce 就好了：

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "query" : {"date" : {"$gt" : week_ago}}})
```

`sort` 选项和 `limit` 一起使用时通常能够发挥非常大的作用。`limit` 也可以单独使用，用来截取一部分文档发送给 `map` 函数。

如果在上个例子中想分析最近 10 000 个页面的访问次数（而不是最近一周的），就可以使用 `limit` 和 `sort`：

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "limit" : 10000, "sort" : {"date" : -1}})
```

`query`、`limit`、`sort` 可以随意组合，但是如果不使用 `limit` 的话，`sort` 就不能有效发挥作用。

4. 使用作用域

MapReduce 可以为 `map`、`reduce`、`finalize` 函数都采用一种代码类型。但多数语言里，可以指定传递代码的作用域。然而 MapReduce 会忽略这个作用域。它有自己的作用域键 `"scope"`，如果想在 MapReduce 中使用客户端的值，则必须使用这个参数。可以用“变量名 : 值”这样的普通文档来设置该选项，然后在 `map`、`reduce` 和 `finalize` 函数中就能使用了。作用域在这些函数内部是不变的。例如，上一节的例子使用 `1/(newDate() - this.date)` 计算页面的新旧程度。可以将当前日期作为作用域的一部分传递进去：

```
> db.runCommand({"mapreduce" : "webpages", "map" : map, "reduce" : reduce,
                 "scope" : {now : new Date()}})
```

这样，在 `map` 函数中就能计算 `1/(now – this.date)` 了。

5. 获得更多的输出

还有个用于调试的详细输出选项。如果想看看 MapReduce 的运行过程，可以将 `"verbose"` 指定为 `true`。

也可以用 `print` 把 `map`、`reduce`、`finalize` 过程中的信息输出到服务器日志上。

7.4 聚合命令

MongoDB 为在集合上执行基本的聚合任务提供了一些命令。这些命令在聚合框架出现之前就已经存在了，现在（大多数情况下）已经被聚合框架取代。然而，复杂的 `group` 操作可能仍然需要使用 JavaScript，`count` 和 `distinct` 操作可以被简化

为普通命令，不需要使用聚合框架。

7.4.1 count

count 是最简单的聚合工具，用于返回集合中的文档数量：

```
> db.foo.count()
0
> db.foo.insert({"x" : 1})
> db.foo.count()
1
```

不论集合有多大，count 都会很快返回总的文档数量。

也可以给 count 传递一个查询文档，Mongo 会计算查询结果的数量：

```
> db.foo.insert({"x" : 2})
> db.foo.count()
2
> db.foo.count({"x" : 1})
1
```

对分页显示来说总数非常必要：“共 439 个，目前显示 0~10 个”。但是，增加查询条件会使 count 变慢。count 可以使用索引，但是索引并没有足够的元数据供 count 使用，所以不如直接使用查询来得快。

7.4.2 distinct

distinct 用来找出给定键的所有不同值。使用时必须指定集合和键。

```
> db.runCommand({"distinct" : "people", "key" : "age"})
```

假设集合中有如下文档：

```
{"name" : "Ada", "age" : 20}
{"name" : "Fred", "age" : 35}
{"name" : "Susan", "age" : 60}
{"name" : "Andy", "age" : 35}
```

如果对 "age" 键使用 distinct，会得到所有不同的年龄：

```
> db.runCommand({"distinct" : "people", "key" : "age"})
{"values" : [20, 35, 60], "ok" : 1}
```

这里还有一个常见问题：有没有办法获得集合里面所有不同的键呢？ MongoDB 并没有直接提供这样的功能，但是可以用 MapReduce（详见 7.3 节）自己写一个。

7.4.3 group

使用group可以执行更复杂的聚合。先选定分组所依据的键，而后MongoDB就会将集合依据选定键的不同值分成若干组。然后可以对每一个分组内的文档进行聚合，得到一个结果文档。



如果你熟悉SQL，那么这个group和SQL中的GROUP BY差不多。

假设现在有个跟踪股票价格的站点。从上午10点到下午4点每隔几分钟就会更新某只股票的价格，并保存在MongoDB中。现在报表程序要获得近30天的收盘价。用group就可以轻松办到。

股价集合中包含数以千计如下形式的文档：

```
{"day" : "2010/10/03", "time" : "10/3/2010 03:57:01 GMT-400", "price" : 4.23}
{"day" : "2010/10/04", "time" : "10/4/2010 11:28:39 GMT-400", "price" : 4.27}
{"day" : "2010/10/03", "time" : "10/3/2010 05:00:23 GMT-400", "price" : 4.10}
{"day" : "2010/10/06", "time" : "10/6/2010 05:27:58 GMT-400", "price" : 4.30}
{"day" : "2010/10/04", "time" : "10/4/2010 08:34:50 GMT-400", "price" : 4.01}
```



注意，由于精度的问题，实际使用中不要将金额以浮点数的方式存储，这个例子只是为了简便才这么做。

我们需要的结果列表中应该包含每天的最后交易时间和价格，就像下面这样：

```
[
    {"time" : "10/3/2010 05:00:23 GMT-400", "price" : 4.10},
    {"time" : "10/4/2010 11:28:39 GMT-400", "price" : 4.27},
    {"time" : "10/6/2010 05:27:58 GMT-400", "price" : 4.30}
]
```

先把集合按照"day"字段进行分组，然后在每个分组中查找"time"值最大的文档，将其添加到结果集中就完成了。整个过程如下所示：

```
> db.runCommand({"group" : {
... "ns" : "stocks",
... "key" : "day",
... "initial" : {"time" : 0},
... "$reduce" : function(doc, prev) {
...     if (doc.time > prev.time) {
...         prev.price = doc.price;
...         prev.time = doc.time;
...     }
... }}})
```

把这个命令分解开看看。

- `"ns" : "stocks"`
  指定要进行分组的集合。

- `"key" : "day"`
  指定文档分组依据的键。这里就是 `"day"` 键。所有 `"day"` 值相同的文档被分到一组。

- `"initial" : {"time" : 0}`
  每一组 `reduce` 函数调用中的初始 `"time"` 值，会作为初始文档传递给后续过程。每一组的所有成员都会使用这个累加器，所以它的任何变化都可以保存下来。

- `"$reduce" : function(doc, prev) { ... }`
  这个函数会在集合内的每个文档上执行。系统会传递两个参数：当前文档和累加器文档（本组当前的结果）。本例中，想让 `reduce` 函数比较当前文档的时间和累加器的时间。如果当前文档的时间更晚一些，则将累加器的日期和价格替换为当前文档的值。别忘了，每一组都有一个独立的累加器，所以不必担心不同日期的命令会使用同一个累加器。

在问题一开始的描述中，就提到只要最近 30 天的股价。然而，我们在这里迭代了整个集合。这就是要添加 `"condition"` 的原因，因为这样就可以只对必要的文档进行处理。

```
> db.runCommand({"group" : {
... "ns" : "stocks",
... "key" : "day",
... "initial" : {"time" : 0},
... "$reduce" : function(doc, prev) {
...     if (doc.time > prev.time) {
...         prev.price = doc.price;
...         prev.time = doc.time;
...     }},
... "condition" : {"day" : {"$gt" : "2010/09/30"}}
... }})
```



有些参考资料提及 `"cond"` 键或者 `"q"` 键，其实和 `"condition"` 键是完全一样的（就是表达力不如 `"condition"` 好）。

最后就会返回一个包含 30 个文档的数组，其实每个文档都是一个分组。每组都包含分组依据的键（这里就是 `"day" : string`）以及这组最终的 `prev` 值。如果有的文档不存在指定用于分组的键，这些文档会被单独分为一组，缺失的键会使用 `"day`

: null"这样的形式。在"condition"中加入"day" : {"$exists" : true}就可以排除不包含指定用于分组的键的文档。group命令同时返回了用到的文档总数和"key"的不同值数量：

```
> db.runCommand({"group" : {...}})
{
    "retval" :
        [
            {
                "day" : "2010/10/04",
                "time" : "Mon Oct 04 2010 11:28:39 GMT-0400 (EST)"
                "price" : 4.27
            },
            ...
        ],
    "count" : 734,
    "keys" : 30,
    "ok" : 1
}
```

这里每组的"price"都是显式设置的，"time"先由初始化器设置，然后在迭代中进行更新。"day"是默认被加进去的，因为用于分组的键会默认加入到每个"retval"内嵌文档中。要是不想在结果集中看到这个键，可以用完成器将累加器文档变为任何想要的形态，甚至变换成非文档（例如数字或字符串）。

1. 使用完成器

完成器（finalizer）用于精简从数据库传到用户的数据，这个步骤非常重要，因为group命令的输出结果需要能够通过单次数据库响应返回给用户。为进一步说明，这里举个博客的例子，其中每篇文章都有多个标签（tag）。现在要找出每天最热门的标签。可以（再一次）按天分组，得到每一个标签的计数。就像下面这样：

```
> db.posts.group({
... "key" : {"day" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
...     }
... }})
```

得到的结果如下所示：

```
[
    {"day" : "2010/01/12", "tags" : {"nosql" : 4, "winter" : 10, "sledding" : 2}},
```

```
    {"day" : "2010/01/13", "tags" : {"soda" : 5, "php" : 2}},
    {"day" : "2010/01/14", "tags" : {"python" : 6, "winter" : 4, "nosql": 15}}
]
```

接着可以在客户端找出 `"tags"` 文档中出现次数最多的标签。然而，向客户端发送每天所有的标签文档需要许多额外的开销——每天所有的键 / 值对都被传送给用户，而我们需要的仅仅是一个字符串。这也就是 `group` 有一个可选的 `"finalize"` 键的原因。`"finalize"` 可以包含一个函数，在每组结果传递到客户端之前调用一次。可以使用 `"finalize"` 函数将不需要的内容从结果集中移除：

```
> db.runCommand({"group" : {
... "ns" : "posts",
... "key" : {"day" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
...     },
... "finalize" : function(prev) {
...     var mostPopular = 0;
...     for (i in prev.tags) {
...         if (prev.tags[i] > mostPopular) {
...             prev.tag = i;
...             mostPopular = prev.tags[i];
...         }
...     }
...     delete prev.tags
... }}})
```

现在，我们就得到了想要的信息，服务器返回的内容可能如下：

```
[
    {"day" : "2010/01/12", "tag" : "winter"},
    {"day" : "2010/01/13", "tag" : "soda"},
    {"day" : "2010/01/14", "tag" : "nosql"}
]
```

`finalize` 可以对传递进来的参数进行修改，也可以返回一个新值。

2. 将函数作为键使用

有时分组所依据的条件可能会非常复杂，而不是单个键。比如要使用 `group` 计算每个类别有多少篇博客文章（每篇文章只属于一个类别）。由于不同作者的风格不同，填写分类名称时可能有人使用大写也有人使用小写。所以，如果要是按类别名来分组，最后“MongoDB”和“mongodb”就是两个完全不同的组。为了消除这种大小

写的影响，就要定义一个函数来决定文档分组所依据的键。

定义分组函数就要用到 $keyf 键（注意不是 "key"），使用 "$keyf" 的 group 命令如下所示：

```
> db.posts.group({"ns" : "posts",
... "$keyf" : function(x) { return x.category.toLowerCase(); },
... "initializer" : ... })
```

有了 "$keyf"，就能依据各种复杂的条件进行分组了。

第 8 章

应用程序设计

本章介绍如何设计应用程序，以便更好地使用 MongoDB，内容包括：

- 内嵌数据和引用数据之间的权衡；
- 优化技巧；
- 数据一致性；
- 模式迁移；
- 不适合使用 MongoDB 作为数据存储的场景。

8.1 范式化与反范式化

数据表示的方式有很多种，其中最重要的问题之一就是在多大程度上对数据进行范式化。范式化（normalization）是将数据分散到多个不同的集合，不同集合之间可以相互引用数据。虽然很多文档可以引用某一块数据，但是这块数据只存储在一个集合中。所以，如果要修改这块数据，只需修改保存这块数据的那一个文档就行了。但是，MongoDB 没有提供连接（join）工具，所以在不同集合之间执行连接查询需要进行多次查询。

反范式化（denormalization）与范式化相反：将每个文档所需的数据都嵌入在文档内部。每个文档都拥有自己的数据副本，而不是所以文档共同引用同一个数据副本。这意味着，如果信息发生了变化，那么所有相关文档都需要进行更新，但是在执行查询时，只需要一次查询，就可以得到所有数据。

决定何时采用范式化何时采用反范式化是比较困难的。范式化能够提高数据写入速度，反范式化能够提高数据读取速度。需要根据自己应用程序的实际需要仔细权衡。

8.1.1 数据表示的例子

假设要保存学生和课程信息。一种表示方式是使用一个 students 集合（每个学生是一个文档）和一个 classes 集合（每门课程是一个文档）。然后用第三个集合 studentClasses 保存学生和课程之间的联系。

```
> db.studentClasses.findOne({"studentId" : id})
{
    "_id" : ObjectId("512512c1d86041c7dca81915"),
    "studentId" : ObjectId("512512a5d86041c7dca81914"),
    "classes" : [
        ObjectId("512512ced86041c7dca81916"),
        ObjectId("512512dcd86041c7dca81917"),
        ObjectId("512512e6d86041c7dca81918"),
        ObjectId("512512f0d86041c7dca81919")
    ]
}
```

如果比较熟悉关系型数据库，可能你之前见过这种类型的表连接，虽然你的每个结果文档中可能只有一个学生和一门课程（而不是一个课程 `"_id"` 列表）。将课程放在数组中，这有点儿 MongoDB 的风格，不过实际上通常不会这么保存数据，因为要经历很多次查询才能得到真实信息。

假设要找到一个学生所选的课程。需要先查找 students 集合找到学生信息，然后查询 studentClasses 找到课程 `"_id"`，最后再查询 classes 集合才能得到想要的信息。为了找出课程信息，需要向服务器请求三次查询。很可能你并不想在 MongoDB 中用这种数据组织方式，除非学生信息和课程信息经常发生变化，而且对数据读取速度也没有要求。

如果将课程引用嵌入在学生文档中，就可以节省一次查询：

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        ObjectId("512512ced86041c7dca81916"),
        ObjectId("512512dcd86041c7dca81917"),
        ObjectId("512512e6d86041c7dca81918"),
        ObjectId("512512f0d86041c7dca81919")
    ]
}
```

`"classes"` 字段是一个数组，其中保存了 John Doe 需要上的课程 `"_id"`。需要找

出这些课程的信息时，就可以使用这些 "_id" 查询 classes 集合。这个过程只需要两次查询。如果数据不需要随时访问也不会随时发生变化（“随时”比“经常”要求更高），那么这种数据组织方式是非常好的。

如果需要进一步优化读取速度，可以将数据完全反范式化，将课程信息作为内嵌文档保存到学生文档的 "classes" 字段中，这样只需要一次查询就可以得到学生的课程信息了：

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        {
            "class" : "Trigonometry",
            "credits" : 3,
            "room" : "204"
        },
        {
            "class" : "Physics",
            "credits" : 3,
            "room" : "159"
        },
        {
            "class" : "Women in Literature",
            "credits" : 3,
            "room" : "14b"
        },
        {
            "class" : "AP European History",
            "credits" : 4,
            "room" : "321"
        }
    ]
}
```

上面这种方式的优点是只需要一次查询就可以得到学生的课程信息，缺点是会占用更多的存储空间，而且数据同步更困难。例如，如果物理学的学分变成了 4 分（不再是 3 分），那么选修了物理学课程的每个学生文档都需要更新，而不只是更新“Physics”文档。

最后，也可以混合使用内嵌数据和引用数据：创建一个子文档数组用于保存常用信息，需要查询更详细信息时通过引用找到实际的文档：

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        {
            "_id" : ObjectId("512512ced86041c7dca81916"),
```

```
                "class" : "Trigonometry"
            },
            {
                "_id" : ObjectId("512512dcd86041c7dca81917"),
                "class" : "Physics"
            },
            {
                "_id" : ObjectId("512512e6d86041c7dca81918"),
                "class" : "Women in Literature"
            },
            {
                "_id" : ObjectId("512512f0d86041c7dca81919"),
                "class" : "AP European History"
            }
        ]
    }
```

这种方式也是不错的选择，因为内嵌的信息可以随着需求的变化进行修改：如果希望在一个页面中包含更多（或者更少）的信息，就可以将更多（或者更少）的信息放在内嵌文档中。

需要考虑的另一个重要问题是，信息更新更频繁还是信息读取更频繁？如果这些数据会定期更新，那么范式化是比较好的选择。如果数据变化不频繁，为了优化更新效率而牺牲读取效率就不值得了。

例如，教科书上介绍范式化的一个例子可能是将用户和用户地址保存在不同的集合中。但是，人们几乎不会改变住址，所以不应该为了这种概率极小的情况（某人改变了住址）而牺牲每一次查询的效率。在这种情景下，应该将地址内嵌在用户文档中。

如果决定使用内嵌文档，更新文档时，需要设置一个定时任务（cron job），以确保所做的每次更新都成功更新了所有文档。例如，我们试图将更新扩散到多个文档，在更新完所有文档之前，服务器崩溃了。需要能够检测到这种问题，并且重新进行未完的更新。

一般来说，数据生成越频繁，就越不应该将这些数据内嵌到其他文档中。如果内嵌字段或者内嵌字段数量是无限增长的，那么应该将这些内容保存在单独的集合中，使用引用的方式进行访问，而不是内嵌到其他文档中。评论列表或者活动列表等信息应该保存在单独的集合中，不应该内嵌到其他文档中。

最后，如果某些字段是文档数据的一部分，那么需要将这些字段内嵌到文档中。如果在查询文档时经常需要将某个字段排除，那么这个字段应该放在另外的集合中，而不是内嵌在当前的文档中。表 8-1 给出了一些指导原则。

表8-1　内嵌数据与引用数据的比较

更适合内嵌	更适合引用
子文档较小	子文档较大
数据不会定期改变	数据经常改变
最终数据一致即可	中间阶段的数据必须一致
文档数据小幅增加	文档数据大幅增加
数据通常需要执行二次查询才能获得	数据通常不包含在结果中
快速读取	快速写入

假如我们有一个用户集合。下面是一些可能需要的字段，以及它们是否应该内嵌到用户文档中。

- 用户首选项（account preferences）
  用户首选项只与特定用户相关，而且很可能需要与用户文档内的其他用户信息一起查询。所以用户首选项应该内嵌到用户文档中。

- 最近活动（recent activity）
  这个字段取决于最近活动增长和变化的频繁程度。如果这是个固定长度的字段（比如最近的 10 次活动），那么应该将这个字段内嵌到用户文档中。

- 好友（friends）
  通常不应该将好友信息内嵌到用户文档中，至少不应该将好友信息完全内嵌到用户文档中。下节会介绍社交网络应用的相关内容。

- 所有由用户产生的内容
  不应该内嵌在用户文档中。

8.1.2　基数

一个集合中包含的对其他集合的引用数量叫做*基数*（cardinality）。常见的关系有一对一、一对多、多对多。假如有一个博客应用程序。每篇*博客文章*（post）都有一个*标题*（title），这是一个一对一的关系。每个*作者*（author）可以有多篇文章，这是一个一对多的关系。每篇文章可以有多个*标签*（tag），每个标签可以在多篇文章中使用，所以这是一个多对多的关系。

在 MongoDB 中，many（多）可以被分拆为两个子分类：many（多）和 few（少）。假如，作者和文章之间可能是一对少的关系：每个作者只发表了为数不多的几篇文章。博客文章和标签可能是多对少的关系：文章数量实际上很可能比标签数量多。博客文章和评论之间是一对多的关系：每篇文章都可以拥有很多条评论。

只要确定了少与多的关系，就可以比较容易地在内嵌数据和引用数据之间进行权衡。通常来说，“少”的关系使用内嵌的方式会比较好，“多”的关系使用引用的方式比较好。

8.1.3 好友、粉丝，以及其他的麻烦事项

> 亲近朋友，远离敌人。

很多社交类的应用程序都需要链接人、内容、粉丝、好友，以及其他一些事物。对于这些高度关联的数据使用内嵌的形式还是引用的形式不容易权衡。这一节会介绍社交图谱数据相关的注意事项。通常，关注、好友或者收藏可以简化为一个发布-订阅系统：一个用户可以订阅另一个用户相关的通知。这样，有两个基本操作需要比较高效：如何保存订阅者，如何将一个事件通知给所有订阅者。

比较常见的订阅实现方式有三种。第一种方式是将内容生产者内嵌在订阅者文档中：

```
{
    "_id" : ObjectId("51250a5cd86041c7dca8190f"),
    "username" : "batman",
    "email" : "batman@waynetech.com"
    "following" : [
        ObjectId("51250a72d86041c7dca81910"),
        ObjectId("51250a7ed86041c7dca81936")
    ]
}
```

现在，对于一个给定的用户文档，可以使用形如 `db.activities.find({"user" : {"$in" : user["following"]}})` 的方式查询该用户感兴趣的所有活动信息。但是，对于一条刚刚发布的活动信息，如果要找出对这条活动信息感兴趣的所有用户，就不得不查询所有用户的 `"following"` 字段了。

另一种方式是将订阅者内嵌到生产生文档中：

```
{
    "_id" : ObjectId("51250a7ed86041c7dca81936"),
    "username" : "joker",
    "email" : "joker@mailinator.com"
    "followers" : [
        ObjectId("512510e8d86041c7dca81912"),
        ObjectId("51250a5cd86041c7dca8190f"),
        ObjectId("512510ffd86041c7dca81910")
    ]
}
```

当这个生产者新发布一条信息时，我们立即就可以知道需要给哪些用户发送通知。

这样做的缺点是，如果需要找到一个用户关注的用户列表，就必须查询整个用户集合。这种方式的优缺点与第一种方式的优缺点正好相反。

同时，这两种方式都存在另一个问题：它们会使用户文档变得越来越大，改变也越来越频繁。通常，"following" 和 "followers" 字段甚至不需要返回：查询粉丝列表有多频繁？如果用户比较频繁地关注某些人或者对一些人取消关注，也会导致大量的碎片。因此，最后的方案对数据进一步范式化，将订阅信息保存在单独的集合中，以避免这些缺点。进行这种程度的范式化可能有点儿过了，但是对于经常发生变化而且不需要与文档其他字段一起返回的字段，这非常有用。对 "followers" 字段做这种范式化是有意义的。

用一个集合来保存发布者和订阅者的关系，其中的文档结构可能如下所示：

```
{
    "_id" : ObjectId("51250a7ed86041c7dca81936"), // 被关注者的 "_id"
    "followers" : [
        ObjectId("512510e8d86041c7dca81912"),
        ObjectId("51250a5cd86041c7dca8190f"),
        ObjectId("512510ffd86041c7dca81910")
    ]
}
```

这样可以使用户文档比较精简，但是需要额外的查询才能得到粉丝列表。由于 "followers" 数组的大小会经常发生变化，所以可以在这个集合上启用 "usePowerOf2Sizes"，以保证 users 集合尽可能小。如果将 followers 集合保存在另一个数据库中，也可以在不过多影响 users 集合的前提下对其进行压缩。

应对威尔 · 惠顿[1]效应

不管使用什么样的策略，内嵌字段只能在子文档或者引用数量不是特别大的情况下有效发挥作用。对于比较有名的用户，可能会导致用于保存粉丝列表的文档溢出。对于这种情况的一种解决方案是在必要时使用“连续的”文档。例如：

```
> db.users.find({"username" : "wil"})
{
    "_id" : ObjectId("51252871d86041c7dca8191a"),
    "username" : "wil",
    "email" : "wil@example.com",
    "tbc" : [
        ObjectId("512528ced86041c7dca8191e"),
        ObjectId("5126510dd86041c7dca81924")
    ]
    "followers" : [
```

注 1：威尔 · 惠顿（Wil Wheaton）：美国演员，曾出演过《星际迷航》，并在《生活大爆炸》中出演 Sheldon 的冤家对头。——译者注

```
            ObjectId("512528a0d86041c7dca8191b"),
            ObjectId("512528a2d86041c7dca8191c"),
            ObjectId("512528a3d86041c7dca8191d"),
            ...
        ]
    }
    {
        "_id" : ObjectId("512528ced86041c7dca8191e"),
        "followers" : [
            ObjectId("512528f1d86041c7dca8191f"),
            ObjectId("512528f6d86041c7dca81920"),
            ObjectId("512528f8d86041c7dca81921"),
            ...
        ]
    }
    {
        "_id" : ObjectId("5126510dd86041c7dca81924"),
        "followers" : [
            ObjectId("512673e1d86041c7dca81925"),
            ObjectId("512650efd86041c7dca81922"),
            ObjectId("512650fdd86041c7dca81923"),
            ...
        ]
    }
```

对于这种情况，需要在应用程序中添加从 `"tbc"`（to be continued）数组中取数据的相关逻辑。

8.2　优化数据操作

如果要优化应用程序，首先必须知道对读写性能进行评估以便找到性能瓶颈。对读取操作的优化通常包括正确使用索引，以及尽可能将所需信息放在单个文档中返回。对写入操作的优化通常包括减少索引数量以及尽可能提高更新效率。

经常需要在写入效率更高的模式与读取效率更高的模式之间权衡，所以必须要知道哪种操作对你的应用程序更重要。这里的影响因素并不只是读取和写入的重要性，也包括读取和写入操作的频繁程度：如果对你的应用程序来说写入操作更加重要，但是为了执行一次写入操作需要进行 1000 次读取操作，那么还是应该首先优化读取速度。

8.2.1　优化文档增长

更新数据时，需要明确更新是否会导致文件体积增长，以及增长程度。如果增长程度是可预知的，可以为文档预留足够的增长空间，这样可以避免文档移动，可以提高写入速度。检查一下填充因子：如果它大约是 1.2 或者更大，可以考虑手动填充。

如果要对文档进行手动填充，可以在创建文档时创建一个占空间比较大的字段，文件创建成功之后再将这个字段移除。这样就提前为文档分配了足够的空间供后续使用。假设有一个餐馆评论的集合，其中的文档如下所示：

```
{
    "_id" : ObjectId(),
    "restaurant" : "Le Cirque",
    "review" : "Hamburgers were overpriced."
    "userId" : ObjectId(),
    "tags" : []
}
```

"tags" 字段会随着用户不断添加标签而增长，应用程序可能经常需要执行这样的更新操作：

```
> db.reviews.update({"_id" : id},
... {"$push" : {"tags" : {"$each" : ["French", "Ǹne dining", "hamburgers"]}}}})
```

如果知道 "tags" 通常不会超过 100 字节，可以手工为文档留出足够的填充空间，这样可以避免更新文档时发生文档移动。如果不为文档预留增长空间，那么每当 "tags" 字段增长时，文档就会被移动。可以在文档最后添加一个大字段（随便用什么名字）进行手工填充，如下所示：

```
{
    "_id" : ObjectId(),
    "restaurant" : "Le Cirque",
    "review" : "Hamburgers were overpriced."
    "userId" : ObjectId(),
    "tags" : [],
    "garbage" : "......................................................"+
        "................................................................"+
        "................................................................"
}
```

可以在第一次插入文档时这么做，也可以在 upsert 时使用 "$setOnInsert" 创建这个字段。

更新文档时，总是用 "$unset" 移除 "garbage" 字段。

```
> db.reviews.update({"_id" : id},
... {"$push" : {"tags" : {"$each" : ["French", "Ǹne dining", "hamburgers"]}}},
... "$unset" : {"garbage" : true}})
```

如果 "garbage" 字段存在，"$unset" 操作符可以将其移除；如果这个字段不存在，"$unset" 操作符什么也不做。

如果文档中有一个字段需要增长，应该尽可能将这个字段放在文档最后的位置

（"garbage" 之前）。这样可以稍微提高一点点的性能，因为如果 "tags" 字段发生了增长，MongoDB 不需要重写 "tags" 后面的字段。

8.2.2　删除旧数据

有些数据只在特定时间内有用：几周或者几个月之后，保留这些数据只是在浪费存储空间。有三种常见的方式用于删除旧数据：使用固定集合，使用 TTL 集合，或者定期删除集合。

最简单的方式是使用固定集合：将集合大小设为一个比较大的值，当集合被填满时，将旧数据从固定集合中挤出。但是，固定集合会对操作造成一些限制，而且在密集插入数据时会大大降低数据在固定集合内的存活期。6.1 节有详细介绍。

第二种方式是使用 TTL 集合，TTL 集合可以更精确地控制删除文档的时机。但是，对于写入量非常大的集合来说这种方式可能不够快：它通过遍历 TTL 索引来删除文档。如果 TTL 集合能够承受足够的写入量，使用 TTL 集合删除旧数据可能是最简单的方式了。6.2 节有详细介绍。

最后一种方法是使用多个集合：例如，每个月的文档单独使用一个集合。每当月份变更时，应用程序就开始使用新月份的集合（初始是个空集合），查询时要对当前月份和之前月份的集合都进行查询。对于 6 个月之前创建的集合，可以直接将其删除。这种方式可以应对任意的操作量，但是对于应用程序来说会比较复杂，因为需要使用动态的集合名称（或者数据库名称），也要动态处理对多个数据库的查询。

8.3　数据库和集合的设计

确定了文档结构之后，接下来就要确定使用什么样的集合或者数据库来保存文档。通常这个过程很简单，但是有一些指导原则需要注意。

通常，具有相近模式的文档应该放在相同的集合中。MongoDB 通常不允许使用多个集合进行数据组合，如果有些文档需要进行集中查询或者聚合，那么这些文档应该放在同一个大集合里。例如，可能有一些结构非常不同的文档，但是如果要对它们进行聚合，就需要让它们位于同一个集合内。

对于数据库来说，最大的问题是锁机制（每个数据库上都有一个读 / 写锁）和存储。每一个数据库，在磁盘上都位于自己的文件中（通常也在单独的文件夹中），这意味着，可以让不同的数据库位于不同的磁盘分卷。所以，你可能希望数据库内的所有项目都拥有相近的“质量”、相近的访问模式，或者相近的访问量。

假设我们有一个拥有多个组件的应用程序：日志组件会创建大量的日志数据（日志数据不是很重要），还要有一个用户集合，以及几个用于保存用户生成数据的集合。用户集合是最有价值的：保证用户数据安全是非常重要的。社交活动数据需要放在一个大流量集合中，它不如用户集合重要，但是比日志集合重要。这个集合主要用于用户通知，所以几乎是一个只插入不更新的集合。

按照重要性进行拆分，最后可能得到三个数据库：logs（日志）、activities（活动）、users（用户）。这样做的好处是，最重要的数据集合的数据量可能最小（例如，用户集合内的数据通常不如日志集合多）。将所有数据集都存储在 SSD 上你可能负担不起，但是也许可以只将用户集合存储在 SSD 上。或者对用户集合使用 RAID10，而对日志和活动集合使用 RAID0。

注意，使用多个数据库时有一些限制：MongoDB 通常不允许直接将数据从一个数据库移到另一个数据库。例如，无法将在 A 数据库上执行 MapReduce 的结果保存到 B 数据库中，也无法使用 `renameCollection` 命令将集合从一个数据库移动到另一个数据库（比如，可以将 foo.bar 重命名为 foo.baz，但是不能将 foo.bar 重命名为 foo2.baz）。

8.4　一致性管理

必须要明确知道应用程序的读取对数据一致性的要求有多高。MongoDB 支持多种不同的一致性级别，从每次都读到完全正确的最新数据到读取不确定新旧程度的数据。如果要得到最近一年内的活动信息报表，可能只要求最近这些天的数据完全准确。相反，如果要做实时交易，可能需要即时读到最新的数据。

要理解如何获得这些不同级别的一致性，首先要了解 MongoDB 的内部机制。服务器为每个数据库连接维护一个请求队列。客户端每次发来的新请求都会添加到队列的末尾。入队之后，这个连接上的请求会依次得到处理。一个连接拥有一个一致的数据库视图，可以总是读取到这个连接最新写入的数据。

注意，每个列队只对应一个连接：如果打开两个 shell，连接到相同的数据库，这时就存在两个不同的连接。如果在其中一个 shell 中执行插入操作，紧接着在另一个 shell 中执行查询操作，新插入的数据可能不会出现在查询结果中。但是，如果是在同一个 shell 中，插入一个文档然后执行查询，一定能够查询到刚插入的文档。想手动重现这种问题是很困难的，但是在一个频繁执行插入和查询的服务器上很可能会发生。经常会有一些开发者使用一个线程插入数据，然后使用另一个线程检查数据是否成功插入。片刻之后，刚刚的数据看上去好像并没有成功插入，但是这些数据忽然就出现了。

使用 Ruby、Python 和 Java 驱动程序时尤其要注意这个问题，因为这三种语言的驱动程序都使用了连接池（connection pool）。为了提高效率，这些驱动程序会建立多个与服务器之间的连接（也就是一个*连接池*），将请求通过不同的连接发送到服务器。但是它们都有各自的机制来保证一系列相关的请求会被同一个连接处理。关于不同语言连接池的详细文档，可以查看 MongoDB Wiki（http://dochub.mongodb.org/drivers/connections）。

当向副本集备份节点（参见第 11 章）发送读取请求时，就更麻烦了。副本集的数据可能不是最新的，这会导致读取到的数据是一秒钟之前或者一分钟之前的，甚至是几个小时之前的。处理这个问题的方式有好几种，最简单的一种是将所有读取请求都发送到主数据库，这样便可以每次都得到最新最准确的数据。也可以设置一个脚本自动检测副本集是否落后于主数据库，如果落后，就将副本集设为维护状态。如果你的副本集比较小，可以使用 `"w" : setSize` 执行安全写入，如果 `getLastError` 没能成功返回，可将后续的读取请求发送到主数据库。

8.5 模式迁移

随着应用程序使用时间的增长和需求变化，数据库模式可能也需要相应地增长和改变。有几种方式可以实现这个需求，不管使用哪种方法，都要小心保存该程序使用过的每一个模式。

最简单的方式就是在应用程序需要时改进数据库模式，以确保应用程序能够支持所有旧版的模式（比如，要能够从容处理某些字段的缺失，或者是某些字段在不同版本中的不同类型）。这种方式可能会导致混乱，尤其是不同版本的模式之间有冲突时。例如，版本 A 要求有 `"mobile"` 字段，但版本 B 没有 `"mobile"` 字段，却需要有另外一个不同字段，同时还有个版本 C 认为 `"mobile"` 字段是可选的。为了满足这样的需求可能会逐步把代码变得一团糟。

另一种稍微结构化一点儿的解决方案是在每个文档中包含一个 `"version"` 字段（或者 `"v"`），使用这个字段来决定应用程序能够接受的文档结构。这种方式对模式的要求更加严格：文档必须对多个版本都有效。这仍然需要支持各种旧版本。

最后一种方式是，当模式发生变化时，将数据进行迁移。通常来说这并不是个好主意：MongoDB 允许使用动态模式，以避免执行迁移，因为执行迁移会对系统造成很大的压力。但是，如果决定改变每一个文档，需要确保所有文档都被成功更新。MongoDB 中的多文档更新并不是原子的（原子是指要么所有文档都成功更新，要么一个也不更新）。如果 MongoDB 在迁移过程中崩溃，最终的结果可能会是只有一部分文档被更新，还有一部分没有更新。

8.6 不适合使用MongoDB的场景

尽管 MongoDB 是一个通用型数据库，可以用在大部分应用程序中，但它并非万能的。MongoDB 不支持下面这些应用场景。

- MongoDB 不支持事务（transaction），对事务性有要求的应用程序不建议使用 MongoDB。可以用几种方式实现简单的类事务（transaction-like）语义，尤其是操作单个文档时，但是数据库并不能强制要求用户这么做。因此，你可以让所有客户端都遵守你设定的某种语义规范（比如，执行任何操作之前都要先检查锁），但是无法阻挡不知情的用户或恶意用户把事情变成一团糟。
- 在多个不同维度上对不同类型的数据进行连接，这是关系型数据库善长的事情。MongoDB 不支持这么做，以后也很可能不支持。
- 最后，如果你使用的工具不支持 MongoDB，那可能你应该选择一个关系型数据库，而不是 MongoDB。有很多工具并不支持 MongoDB，从 SQLAl chemy 到 Wordpress。支持 MongoDB 的工具已经越来越多了，但是目前来说仍然不如关系型数据库多。

第三部分

复制

第9章

创建副本集

本章介绍 MongoDB 的复制系统：副本集（replica set）。本章主要内容如下：

- 副本集的概念；
- 副本集的创建方法；
- 副本集成员的可用选项 。

9.1 复制简介

从第 1 章开始，我们使用的一直是单台服务器，一个 mongod 服务器进程。如果只是用作学习和开发，这是可以的，但是如果用到生产环境中，风险会很高：如果服务器崩溃了或者不可访问了怎么办？数据库至少会有一段时间不可用。如果是硬件出了问题，可能需要将数据转移到另一个机器上。在最坏的情况下，磁盘或者网络问题可能会导致数据损坏或者数据不可访问。

使用复制可以将数据副本保存到多台服务器上，建议在所有的生产环境中都要使用。使用 MongoDB 的复制功能，即使一台或多台服务器出错，也可以保证应用程序正常运行和数据安全。

在 MongoDB 中，创建一个副本集之后就可以使用复制功能了。副本集是一组服务器，其中有一个主服务器（primary），用于处理客户端请求；还有多个备份服务器（secondary），用于保存主服务器的数据副本。如果主服务器崩溃了，备份服务器会自动将其中一个成员升级为新的主服务器。

使用复制功能时，如果有一台服务器宕机了，仍然可以从副本集的其他服务器上访问数据。如果服务器上的数据损坏或者不可访问，可以从副本集的某个成员中创建一份新的数据副本。

本章主要介绍副本集以及如何在系统上建立复制功能。

9.2　建立副本集

为了快速入门，本节会指导你在本地机器上建立一个包含三个成员的副本集。这些设置不适用于生产环境，但是可以让你熟悉复制功能以及相关的各种配置。



本节例子中的数据保存在 /data/db 目录下，应该在运行这些代码之前确保这个目录存在，而且当前用户对这个目录拥有写权限。

使用 `--nodb` 选项启动一个 mongo shell，这样可以启动 shell 但是不连接到任何 mongod：

```
$ mongo --nodb
```

通过执行下面的命令就可以创建一个副本集：

```
> replicaSet = new ReplSetTest({"nodes" : 3})
```

这行代码可以创建一个包含三个服务器的副本集：一个主服务器和两个备份服务器。但是，在执行下面两个命令之前 mongod 服务器不会真正启动：

```
> // 启动 3 个 mongod 进程
> replicaSet.startSet()
>
> // 配置复制功能
> replicaSet.initiate()
```

现在已经有了 3 个 mongod 进程，分别运行在 31000、31001 和 31002 端口。这 3 个进程都会把各自的日志输出到当前 shell 中，这会让人很混乱。所以先把这个 shell 放在一边，再开启一个新的 shell 用于工作吧。

在第二个 shell 中，连接到运行在 31000 端口的 mongod：

```
> conn1 = new Mongo("localhost:31000")
connection to localhost:31000
testReplSet:PRIMARY>
testReplSet:PRIMARY> primaryDB = conn1.getDB("test")
test
```

注意，当连接到一个副本集成员时，提示符变成了 "testReplSet:PRIMARY>"。其中 "PRIMARY" 是当前成员的状态，"testReplSet" 是副本集的标识符。"testReplSet" 是 ReplSetTest 使用的默认名称，之后会讲述如何自定义副本集标识符。

为了简洁和可读性，之后的例子会使用 ">" 代替 "testReplSet:PRIMARY>" 提示符。

在连接到主节点的连接上执行 isMaster 命令，可以看到副本集的状态：

```
> primaryDB.isMaster()
{
    "setName" : "testReplSet",
    "ismaster" : true,
    "secondary" : false,
    "hosts" : [
        "wooster:31000",
        "wooster:31002",
        "wooster:31001"
    ],
    "primary" : "wooster:31000",
    "me" : "wooster:31000",
    "maxBsonObjectSize" : 16777216,
    "localTime" : ISODate("2012-09-28T15:48:11.025Z"),
    "ok" : 1
}
```

isMaster 返回的字段有点儿多，其中有一个很重要的字段指明了这是一个主节点（"ismaster" : true），副本集中还有一个 hosts 列表。



如果服务器返回内容 "ismaster" : false，也是正常的。可以从 "primary" 字段获知主节点是哪一个，然后重新连接到主节点所在的主机/端口就可以了。

既然已经连接到主节点，就做一些写入操作看看会有什么发生吧！首先，插入 1000 个文档：

```
> for (i=0; i<1000; i++) { primaryDB.coll.insert({count: i}) }
>
> // 检查集合的文档数量，确保真的插入成功了
> primaryDB.coll.count()
1000
```

检查其中一个副本集成员，验证一下其中是否有刚刚写入的那些文档的副本。可以连接到任意一个备份节点：

```
> conn2 = new Mongo("localhost:31001")
connection to localhost:31001
> secondaryDB = conn2.getDB("test")
test
```

备份节点可能会落后于主节点，可能没有最新写入的数据，所以备份节点在默认情况下会拒绝读取请求，以防止应用程序意外拿到过期的数据。因此，如果在备份节点上做查询，可能会得到一个错误提示，说当前节点不是主节点。

```
> secondaryDB.coll.find()
error: { "$err" : "not master and slaveok=false", "code" : 13435 }
```

这是为了保护应用程序，以免意外连接到备份节点，读取到过期数据。如果希望从备份节点读取数据，需要设置“从备份节点读取数据没有问题”标识，如下所示：

```
> conn2.setSlaveOk()
```

注意，slaveOk 是对连接（例子中是 conn2）设置的，不是对数据库（secondaryDB）设置的。

现在就可以从这个备份节点中读取数据了。使用普通的查询：

```
> secondaryDB.coll.find()
{ "_id" : ObjectId("5037cac65f3257931833902b"), "count" : 0 }
{ "_id" : ObjectId("5037cac65f3257931833902c"), "count" : 1 }
{ "_id" : ObjectId("5037cac65f3257931833902d"), "count" : 2 }
...
{ "_id" : ObjectId("5037cac65f3257931833903c"), "count" : 17 }
{ "_id" : ObjectId("5037cac65f3257931833903d"), "count" : 18 }
{ "_id" : ObjectId("5037cac65f3257931833903e"), "count" : 19 }
Type "it" for more
>
> secondaryDB.coll.count()
1000
```

可以看到刚刚写入的所有文档都出现在备份节点中了。

现在，试着在上执行写入操作：

```
> secondaryDB.coll.insert({"count" : 1001})
> secondaryDB.runCommand({"getLastError" : 1})
{
    "err" : "not master",
    "code" : 10058,
    "n" : 0,
    "lastOp" : Timestamp(0, 0),
    "connectionId" : 5,
    "ok" : 1
}
```

可以看到，不能对备份节点执行写操作。备份节点只通过复制功能写入数据，不接受客户端的写入请求。

有一个很有意思的功能你应该试一下：自动故障转移（automatic failover）。如果主节点挂了，其中一个备份节点会自动选举为主节点。为了验证这个功能，先关掉主节点：

```
> primaryDB.adminCommand({"shutdown" : 1})
```

在备份节点上执行 `isMaster`，看看新的主节点是哪一个：

```
> secondaryDB.isMaster()
```

返回的内容如下所示：

```
{
    "setName" : "testReplSet",
    "ismaster" : true,
    "secondary" : false,
    "hosts" : [
        "wooster:31001",
        "wooster:31000",
        "wooster:31002"
    ],
    "primary" : "wooster:31001",
    "me" : "wooster:31001",
    "maxBsonObjectSize" : 16777216,
    "localTime" : ISODate("2012-09-28T16:52:07.975Z"),
    "ok" : 1
}
```

新的主节点也可以是其他服务器。第一个检测到主节点挂了的备份节点会成为新的主节点。现在可以向新的主节点发送写入请求了。

`isMaster` 是一个非常老的命令了，那时副本集还没有出现，MongoDB 只支持主从复制（master-slave replication）。所以它与副本集的术语有些不一致，`isMaster` 中的主节点（master）与副本集中的主节点（primary）是等同的，从节点（slave）则相当于备份节点（secondary）。

在副本集上完成这些操作之后，从第一个 shell 中将其关闭。这个 shell 中现在应该充满了大量的副本集成员输出日志，敲几次 Enter 键之后就可以看到命令提示符了。可以执行下面的命令关闭副本集：

```
> replicaSet.stopSet()
```

恭喜！你刚刚已经完成了创建副本集、使用副本集和关闭副本集的操作！

有几个关键的概念需要注意。

- 客户端在单台服务器上可以执行的请求，都可以发送到主节点执行（读、写、执行命令、创建索引等）。
- 客户端不能在备份节点上执行写操作。
- 默认情况下，客户端不能从备份节点中读取数据。在备份节点上显式地执行 `setSlaveOk` 之后，客户端就可以从备份节点中读取数据了。

理解这些基本知识之后，本章剩余的部分是集中讲述在各种实际情况下应该如何配置副本集。记住，如果希望在实际中看看某个配置或者选项的效果，随时可以回到 `ReplSetTest`。

9.3 配置副本集

在实际的部署中，需要在多台机器之间建立复制功能。本节会完整建立一个真实场景下的副本集，你在自己的应用程序中可以直接使用。

假设你有一个运行在 server-1:27017 上的单个 mongod 实例，其中已经有一些数据（如果数据库中现在没有数据也没关系，只是数据目录会为空而已）。首先要为副本集选定一个名字，名字可以是任意的 UTF-8 字符串。

选好名称之后，使用 `--replSet name` 选项重启 `server-1`。例如：

```
$ mongod --replSet spock -f mongod.conf --fork
```

现在，使用同样的 `replSet` 和标示符（`spock`）再启动两个 mongod 服务器作为副本集中的其他成员：

```
$ ssh server-2
server-2$ mongod --replSet spock -f mongod.conf --fork
server-2$ exit
$
$ ssh server-3
server-3$ mongod --replSet spock -f mongod.conf --fork
server-3$ exit
```

只有第一个副本集成员拥有数据，其他成员的数据目录都是空的。只要将后两个成员添加到副本集中，它们就会自动克隆第一个成员的数据。

将 `replSet` 选项添加到每个成员各有自的 mongod.conf 文件中，以后启动时就会自动使用这个选项。

现在应该有 3 个分别运行在不同服务器上的 mongod 实例了。但是，每个 mongod

都不知道有其他 mongod 存在。为了让每个 mongod 能够知道彼此的存在，需要创建一个配置文件，在配置文件中列出每一个成员，并且将配置文件发送给 server-1，然后 server-1 会负责将配置文件传播给其他成员。

首先创建配置文件。在 shell 中，创建一个如下所示的文档：

```
> config = {
    "_id" : "spock",
    "members" : [
        {"_id" : 0, "host" : "server-1:27017"},
        {"_id" : 1, "host" : "server-2:27017"},
        {"_id" : 2, "host" : "server-3:27017"}
    ]
}
```

这个配置文档中有几个重要的部分。"_id" 字段的值就是启动时从命令行传递进来的副本集名称（在本例中是 "spock"）。一定要保证这个名称与启动时传入的名称一致。

这个文档的剩余部分是一个副本集成员数组。其中每个元素都需要两个字段：一个唯一的数值类型的 "_id" 字段，和一个主机名（将例子中的主机名替换为你自己实际使用的主机地址）。

这个 config 对象就是副本集的配置，现在需要将其发送给其中一个副本集成员。为此，连接到一个有数据的服务器（server-1:27017），使用 config 对象对副本集进行初始化：

```
> // 连接到 server-1
> db = (new Mongo("server-1:27017")).getDB("test")
>
> // 初始化副本集
> rs.initiate(config)
{
    "info" : "Config now saved locally. Should come online in about a minute.",
    "ok" : 1
}
```

server-1 会解析这个配置对象，然后向其他成员发送消息，提醒它们使用新的配置。所有成员都配置完成之后，它们会自动选出一个主节点，然后就可以正常处理读写请求了。



可惜，无法将单机服务器转换为副本集，除非停机重启并进行初始化。即使只有一个服务器，可能你也想将它配置为一个只有一个成员的副本集。有了这样一个副本集之后，继续添加更多的成员时就不需要停机了。

如果正在创建一个全新的副本集，可以将配置文件发送给副本集的任何一个成员。如果副本集中已经有一个有数据的成员，那就必须将配置对象发送给这个拥有数据的成员。如果拥有数据的成员不止一个，那么就无法初始化副本集。



必须使用 mongo shell 来配置副本集。没有其他方法可以基于文件对副本集进行配置。

9.3.1 rs辅助函数

注意上面的 `rs.initiate()` 命令中的 `rs`。`rs` 是一个全局变量，其中包含与复制相关的辅助函数（可以执行 `rs.help()` 查看可用的辅助函数）。这些函数大多只是数据库命令的包装器。例如，下面的数据库命令与 `rs.initiate(conǸg)` 是等价的：

```
> db.adminCommand({"replSetInitiate" : config})
```

对辅助函数和底层的数据库命令都做些了解是非常好的，有时直接使用数据库命令比使用辅助函数要简单。

9.3.2 网络注意事项

副本集内的每个成员都必须能够连接到其他所有成员（包括自身）。如果遇到某些成员不能到达其他运行中成员的错误，就需要更改网络配置以便各个成员能够相互连通。

另外，副本集的配置中不应该使用 localhost 作为主机名。如果所有副本集成员都运行在同一台机器上，那么 localhost 可以被正确解析，但是运行在一台机器上的副本集意义不大；如果副本集是运行在多台机器上的，那么 localhost 就无法被解析为正确的主机名。MongoDB 允许副本集的所有成员都运行在同一台机器上，这样可以方便在本地测试，但是如果在配置中混用 localhost 和非 localhost 主机名的话，MongoDB 会给出警告。

9.4 修改副本集配置

可以随时修改副本集的配置：可以添加或者删除成员，也可以修改已有的成员。很多常用操作都有对应的 shell 辅助函数，比如，可以使用 `rs.add` 为副本集添加新成员：

```
> rs.add("server-4:27017")
```

类似地，也可以从副本集中删除成员：

```
> rs.remove("server-1:27017")
Fri Sep 28 16:44:46 DBClientCursor::init call() failed
Fri Sep 28 16:44:46 query failed : admin.$cmd { replSetReconfig: {
    _id: "testReplSet", version: 2, members: [ { _id: 0, host: "ubuntu:31000" },
    { _id: 2, host: "ubuntu:31002" } ] } } to: localhost:31000
Fri Sep 28 16:44:46 Error: error doing query:
    failed src/mongo/shell/collection.js:155
Fri Sep 28 16:44:46 trying reconnect to localhost:31000
Fri Sep 28 16:44:46 reconnect localhost:31000 ok
```

注意，删除成员时（或者是除添加成员之外的其他改变副本集配置的行为），会在 shell 中得到很多无法连接数据库的错误信息。这是正常的，这实际上说明配置修改成功了。重新配置副本集时，作为重新配置过程的最后一步，主节点会关闭所有连接。因此，shell 中的连接会短暂断开，然后重新自动建立连接。

重新配置副本集时，主节点需要先退化为普通的备份节点，以便接受新的配置，然后会恢复。要注意，重新配置副本集之后会，副本集中会暂时没有主节点，之后会一切恢复正常。

可以在 shell 中执行 `rs.config()` 来查看配置修改是否成功。这个命令可以打印出副本集当前使用的配置信息：

```
> rs.config()
{
    "_id" : "testReplSet",
    "version" : 2,
    "members" : [
        {
            "_id" : 1,
            "host" : "server-2:27017"
        },
        {
            "_id" : 2,
            "host" : "server-3:27017"
        },
        {
            "_id" : 3,
            "host" : "server-4:27017"
        }
    ]
}
```

每次修改副本集配置时，`"version"` 字段都会自增，它的初始值为 1。

除了对副本集添加或者删除成员，也可以修改现有的成员。为了修改副本集成员，可以在 shell 中创建新的配置文档，然后调用 `rs.reconǸg`。假设有如下所示的配置：

```
> rs.config()
{
    "_id" : "testReplSet",
    "version" : 2,
    "members" : [
        {
            "_id" : 0,
            "host" : "server-1:27017"
        },
        {
            "_id" : 1,
            "host" : "10.1.1.123:27017"
        },
        {
            "_id" : 2,
            "host" : "server-3:27017"
        }
    ]
}
```

其中 `"_id"` 为 1 的成员地址用 IP 而不是主机名表示，需要将其改为主机名表示的地址。首先在 shell 中得到当前使用的配置，然后修改相应的字段：

```
> var config = rs.config()
> config.members[1].host = "server-2:27017"
```

现在配置文件修改完成了，需要使用 `rs.reconfig` 辅助函数将新的配置文件发送给数据库：

```
> rs.reconfig(config)
```

对于复杂的数据集配置修改，`rs.reconfig` 通常比 `rs.add` 和 `rs.remove` 更有用，比如修改成员配置或者是一次性添加或者删除多个成员。可以使用这个命令做任何合法的副本集配置修改：只需创建想要的配置文档然后将其传给 `rs.reconfig`。

9.5 设计副本集

为了能够设计自己的副本集，有一些特定的副本集相关概念需要熟悉。下一章会详细讲述这些内容。副本集中很重要的一个概念是“大多数”（majority）：选择主节点时需要由大多数决定，主节点只有在得到大多数支持时才能继续作为主节点，写操作被复制到大多数成员时这个写操作就是安全的。这里的*大多数*被定义为“副本中一半以上的成员”，如表 9-1 所示。

注意，如果副本集中有些成员挂了或者是不可用，并不会影响“大多数”。因为“大多数”是基于副本集的配置来计算的。

表9-1　怎样才算大多数

副本集中的成员总数	副本集中的大多数
1	1
2	2
3	2
4	3
5	3
6	4
7	4

假设有一个包含 5 个成员的副本集，其中 3 个成员不可用，仍然有 2 个可以正常工作，如图 9-1 所示。剩余的 2 个成员已经无法达到副本集“大多数”的要求（在这个例子中，至少要有 3 个成员才算“大多数”），所以它们无法选举主节点。如果这两个成员中有一个是主节点，当它注意到它无法得到“大多数”成员支持时，就会从主节点上退位。几秒钟之后，这个副本集中会包含 2 个备份节点和 3 个不可达成员。

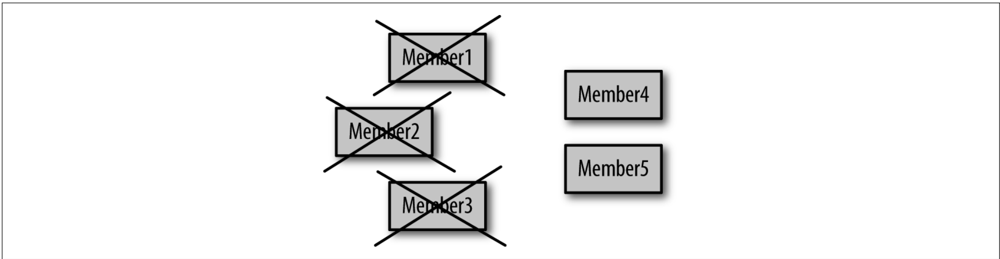


图 9-1：由于副本集中只有少数成员可用，所有成员都会变为备份节点

可能会有很多人觉得这样的规则弱爆了：为什么剩余的两个成员不能选举出主节点呢？问题在于，3 个不可达的成员并不一定是真的挂了，可能只是由于网络问题造成不可达，如图 9-2 所示。在这种情况下，左边的 3 个成员可以选举出一个主节点，因为 3 个成员可以达到副本集成员的大多数（总共 5 个成员）。

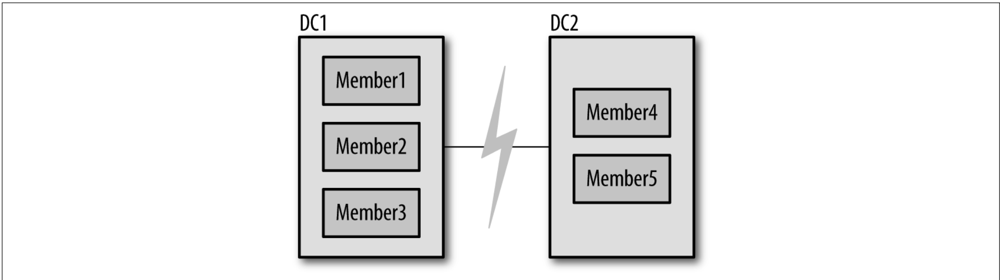


图 9-2：对于成员来说，左边的服务器会觉得右边的服务器挂了，右边的服务器也会觉得左边的服务器挂了

在这种情况下，我们不希望两边的网络各自选举出一个主节点：那样的话副本集就会拥有两个主节点了！两个主节点都可以写入数据，这样整个副本集的数据就会发生混乱。只有达到“大多数”的情况下才能选举或者维持主节点，这样要求是为了避免出现多个主节点。

通常只能有一个主节点，这对于副本集的配置是很重要的。例如，对于上面描述的 5 个成员来说，如果 1、2、3 位于同一个数据中心，而 4、5 位于另一个数据中心。这样，在第 1 个数据中心里，几乎总是可以满足“大多数”这个条件（这样就可以比较容易地判断出很可能是数据中心之间的网络错误，而不是数据中心内部的错误）。

一种常见的设置是使用 2 个成员的副本集（这通常不是你想要的）：一个主节点和一个备份节点。假如其中一个成员不可用，另一个成员就看不到它了，如图 9-3 所示。在这种情况下，网络任何一端都无法达到“大多数”的条件，所以这个副本集会退化为拥有两个备份节点（没有主节点）的副本集。因此，通常不建议使用这样的配置。

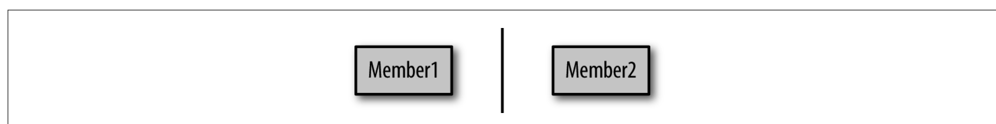


图 9-3：如果成员总数是偶数，成员平均分配到不同的网络中，任何一边都无法满足“大多数”的条件

下面是两种推荐的配置方式。

- 将“大多数”成员放在同一个数据中心，如图 9-2 所示。如果有一个主数据中心，而且你希望副本集的主节点总是位于主数据中心的话，这样的配置会比较好。只要主数据中心能够正常运转，就会有一个主节点。但是，如果主数据中心不可用了，那么备份数据中心的成员无法选举出主节点。
- 在两个数据中心各自放置数量相等的成员，在第三个地方放置一个用于决定胜负的副本集成员。如果两个数据中心同等重要，那么这种配置会比较好。因为任意一个数据中心的服务器都可以找到另一台服务器以达到“大多数”。但是，这样就需要将服务器分散到三个地方。

更复杂的需求需要使用不同的配置，一定要考虑清楚，出现不利情况时，副本集要如何达到“大多数”的要求。

如果 MongoDB 的一个副本集可以拥有多个主节点，上面这些复杂问题就迎刃而解了。但是，多个主节点会带来其他的复杂性。拥有两个主节点的情况下，就需要处理写入冲突（例如，A 在第一个主节点上更新了一个文档，而 B 在另一个主节点上

删除了这个文档）。在支持多线程写入的系统中有两种常见的冲突处理方式：手工解决冲突或者是让系统任选一个作为“赢家”。但是这两种方式对于开发者来说都不容易实现，因为无法确保写入的数据不会被其他节点修改。因此，MongoDB 选择只支持单一主节点。这样可以使开发更容易，但是当副本集被设为只读时，将导致程序暂时无法写入数据。

选举机制

当一个备份节点无法与主节点连通时，它就会联系并请求其他的副本集成员将自己选举为主节点。其他成员会做几项理性的检查：自身是否能够与主节点连通？希望被选举为主节点的备份节点的数据是否最新？有没有其他更高优先级的成员可以被选举为主节点？

如果要求被选举为主节点的成员能够得到副本集中“大多数”成员的投票，它就会成为主节点。即使“大多数”成员中只有一个否决了本次选举，选举就会取消。如果成员发现任何原因，表明当前希望成为主节点的成员不应该成为主节点，那么它就会否决此次选举。

在日志中可以看到得票数为比较大的负数的情况，因为一张否决票相当于 10 000 张赞成票。如果某个成员投赞成票，另一个成员投否决票，那么就可以在消息中看到选举结果为 –9999 或者是比较相近的负数值。

```
Wed Jun 20 17:44:02 [rsMgr] replSet info electSelf 1
Wed Jun 20 17:44:02 [rsMgr] replSet couldn't elect self, only received -9999 votes
```

如果有两个成员投了否决票，一个成员投了赞成票，那么选举结果就是 –19999，依次类推。这些消息是很正常的，不必担心。

希望成为主节点的成员（候选人）必须使用复制将自己的数据更新为最新，副本集中的其他成员会对此进行检查。复制操作是严格按照时间排序的，所以候选人的最后一条操作要比它能连通的其他所有成员更晚（或者与其他成员相等）。

假设候选人执行的最后一个复制操作是 123。它能连通的其他成员中有一个的最后复制操作是 124，那么这个成员就会否决候选人的选举。这时候选人会继续进行数据同步，等它同步到 124 时，它会重新请求选举（如果那时整个副本集中仍然没有主节点的话）。在新一轮的选举中，假如候选人没有其他不合规之处，之前否决它的成员就会为它投赞成票。

假如候选人得到了“大多数”的赞成票，它就会成为主节点。

还有一点需要注意：每个成员都只能要求自己被选举为主节点。简单起见，不能推荐其他成员被选举为主节点，只能为申请成为主节点的候选人投票。

9.6 成员配置选项

到目前为止，我们建立的副本集中所有成员都拥有同样的配置。但是，有时我们并不希望每个成员都完全一样。你可能希望让某个成员拥有优先成为主节点的权力，或者是让某个成员对客户端不可见，这样便不会有读写请求发送给它。在副本集配置的子文档中可以为每个成员指定这些选项（甚至更多选项）。本节介绍可以对成员使用的选项。

9.6.1 选举仲裁者

上面的例子显示了具有两个成员的副本集在"大多数"要求上的缺点。但是，很多人的应用程序使用量比较小，并不想保存三份数据副本。两份副本已经足够了，保存第三份副本的话纯粹是浪费人力、物力和财力。

对于这种部署，MongoDB 支持一种特殊类型的成员，称为*仲裁者*（arbiter）。仲裁者的唯一作用就是参与选举。仲裁者并不保存数据，也不会为客户端提供服务：它只是为了帮助具有两个成员的副本集能够满足"大多数"这个条件。

由于仲裁者并不需要履行传统 mongod 服务器的责任，所以可以将仲裁者作为轻量级进程，运行在配置比较差的服务器上。如果可能，应该将仲裁者放在单独的故障域（failure domain）中，与其他成员分开。这样它就可以以"外部视角"来看待副本集中的成员了，如 9.5 节在部署列表中推荐的一样。

启动仲裁者与启动普通 mongod 的方式相同，使用 "`--replSet` 副本集名称 " 和空的数据目录。可以使用 `rs.addArb()` 辅助函数将仲裁者添加到副本集中：

```
> rs.addArb("server-5:27017")
```

也可以在成员配置中指定 `arbiterOnly` 选项，这与上面的效果是一样的：

```
> rs.add({"_id" : 4, "host" : "server-5:27017", "arbiterOnly" : true})
```

成员一旦以仲裁者的身份添加到副本集中，它就永远只能是仲裁者：无法将仲裁者重新配置为非仲裁者，反之亦然。

使用仲裁者的另一个好处是：如果你拥有的节点数是偶数，那么可能会出现一半节点投票给 A，但是另一半成员投票给 B 的情况。仲裁者这时就可以投出决定胜负的

关键一票。

1. 最多只能使用一个仲裁者

注意，在上面的例子中，最多只需要一个仲裁者。如果节点数量是奇数，那就不需要仲裁者。一种错误的理解是：为了“以防万一”，总是应该添加额外的仲裁者。但是，添加额外的仲裁者，并不能加快选举速度，也不能提供更好的数据安全性。

假设有一个 3 成员的副本集。需要两个成员才能组成“大多数”，才能选举主节点。如果这时添加了一个仲裁者，副本集中总共就有 4 个成员了，要有 3 个成员才能组成“大多数”。因此，副本集的稳定性其实是降低了：原本只需要 67% 的成员可用，副本集就可用；现在必须要有 75% 的成员可用，副本集才可用。

添加额外成员也会导致选举耗时变长。由于添加了仲裁者，现在副本集一共拥有偶数个成员，这样就可能出现两个成员票数相同的情况。仲裁者的目的应该是避免出现平票，而不是导致出现平票。

2. 仲裁者的缺点

不知道应该将一个成员作为数据节点还是作为仲裁者时，应该将其作为数据节点。在小副本集中使用仲裁者而不是数据节点会导致一些操作性的任务变困难。假设有一个副本集，它有两个“普通”成员，还有一个仲裁者成员，其中一个数据成员挂了。如果这个数据成员真的挂了（数据无法恢复），另一个数据成员成为主节点。这时整个副本集中只有一个数据成员和一个仲裁者成员。为了保证数据安全，就需要一个新的备份节点，并且将主节点的数据副本复制到备份节点。复制数据会对服务器造成很大的压力，会拖慢应用程序。通常，将几 GB 的数据复制到新服务器可以很快完成，不会对服务器和应用程序造成显著影响，但是如果要复制 100 GB 以上的数据，问题就会很严重了。

相反，如果拥有三个数据成员，一个服务器挂掉时，副本集中仍然有一个主节点和一个备份节点，不会影响正常运作。这时，可以用剩余的那个备份节点来初始化一个新的备份节点服务器，而不必依赖于主节点。

在上面两个数据成员 + 一个仲裁者成员的情景中，主节点是仅剩的一份完好的数据，它不仅要处理应用程序请求，还要将数据复制到另一个新的服务器上。

如果可能，尽可能在副本集中使用奇数个数据成员，而不要使用仲裁者。

9.6.2　优先级

优先级用于表示一个成员渴望成为主节点的程度。优先级的取值范围可以是 0~100，

默认是 1。将优先级设为 0 有特殊含义：优先级为 0 的成员永远不能够成为主节点。这样的成员称为被动成员（passive member）。

拥有最高优先级的成员会优先选举为主节点（只要它能够得到集合中“大多数”的赞成票，并且数据是最新的）。假如在副本集中添加了一个优先级为 1.5 的成员：

```
> rs.add({"_id" : 4, "host" : "server-4:27017", "priority" : 1.5})
```

假设其他成员的优先级都是 1，只要 server-4 拥有最新的数据，那么当前的主节点就会自动退位，server-4 会被选举为新的主节点。如果 server-4 的数据不够新，那么当前主节点就会保持不变。设置优先级并不会导致副本集中选不出主节点，也不会使数据不够新的成员成为主节点（一直到它的数据更新到最新）。

使用优先级时有一点需要注意：修改副本集配置时，新的配置必须要发送给在新配置下可能成为主节点的成员。因此，无法在一次 `reconfig` 操作中将当前主节点的优先级设置为 0，也不能对所有成员优先级都为 0 的副本集执行 `reconfig`。

优先值的值只会影响副本集成员间相对优先级大小关系。如果某个副本集 3 个成员的优先级是 500、1、1，另一个副本集 3 个成员的优先级是 2、1、1，那么它们的行为是一样的。

9.6.3　隐藏成员

客户端不会向隐藏成员发送请求，隐藏成员也不会作为复制源（尽管当其他复制源不可用时隐藏成员也会被使用）。因此，很多人会将不够强大的服务器或者备份服务器隐藏起来。

假设有一副本集如下所示：

```
> rs.isMaster()
{
    ...
    "hosts" : [
        "server-1:27107",
        "server-2:27017",
        "server-3:27017"
    ],
    ...
}
```

为了隐藏 server-3，可以在它的配置中指定 `hidden : true`。只有优先级为 0 的成员才能被隐藏（不能将主节点隐藏）：

```
> var config = rs.config()
```

```
> config.members[2].hidden = 0
0
> config.members[2].priority = 0
0
> rs.reconfig(config)
```

现在，执行 isMaster() 可以看到：

```
> rs.isMaster()
{
    ...
    "hosts" : [
        "server-1:27107",
        "server-2:27017"
    ],
    ...
}
```

使用 rs.status() 和 rs.conﬁg() 能够看到隐藏成员，隐藏成员只对 isMaster() 不可见。客户端连接到副本集时，会调用 isMaster() 来查看可用成员。因此，隐藏成员不会收到客户端的读请求。

要将隐藏成员设为非隐藏，只需将配置中的 hidden 设为 false 就可以了，或者删除 hidden 选项。

9.6.4　延迟备份节点

数据可能会因为人为错误而遭受毁灭性的破坏：可能有人不小心删除了主数据库，或者刚上线的新版应用程序有一个严重 bug，把所有数据都变成了垃圾。为了防止这类问题，可以使用 slaveDelay 设置一个延迟的备份节点。

延迟备份节点的数据会比主节点延迟指定的时间（单位是秒），这是有意为之。这样，如果有人不小心摧毁了你的主集合，还可以将数据从先前的备份中恢复过来。12.4.7 节有详细介绍。

slaveDelay 要求成员的优先级是 0。如果你的应用会将读请求路由到备份节点，应该将延迟备份节点隐藏掉，以免读请求被路由到延迟备份节点。

9.6.5　创建索引

有时，备份节点并不需要与主节点拥有相同的索引，甚至可以没有索引。如果某个备份节点的用途仅仅是处理数据备份或者是离线的批量任务，那么你可能希望在它的成员配置中指定 "buildIndexs" : false。这个选项可以阻止备份节点创建索引。

这是一个永久选项，指定了 `"buildIndexes" : false` 的成员永远无法恢复为可以创建索引的“正常”成员。如果确实需要将不创建索引的成员修改为可以创建索引的成员，那么必须将这个成员从副本集中移除，再删除它的所有数据，最后再将它重新添加到副本集中，并且允许它重新进行数据同步。

另外，这个选项也要求成员的优先级为 0。

第10章
副本集的组成

本章介绍副本集的各个部分是如何组织在一起的，包括：

- 副本集成员如何复制新数据；
- 如何让新成员开始工作；
- 选举机制；
- 可能的服务器和网络故障。

10.1 同步

复制用于在多台服务器之间备份数据。MongoDB 的复制功能是使用操作日志 oplog 实现的，操作日志包含了主节点的每一次写操作。oplog 是主节点的 local 数据库中的一个固定集合。备份节点通过查询这个集合就可以知道需要进行复制的操作。

每个备份节点都维护着自己的 oplog，记录着每一次从主节点复制数据的操作。这样，每个成员都可以作为同步源提供给其他成员使用，如图 10-1 所示。备份节点从当前使用的同步源中获取需要执行的操作，然后在自己的数据集上执行这些操作，最后再将这些操作写入自己的 oplog。如果遇到某个操作失败的情况（只有当同步源的数据损坏或者数据与主节点不一致时才可能发生），那么备份节点就会停止从当前的同步源复制数据。

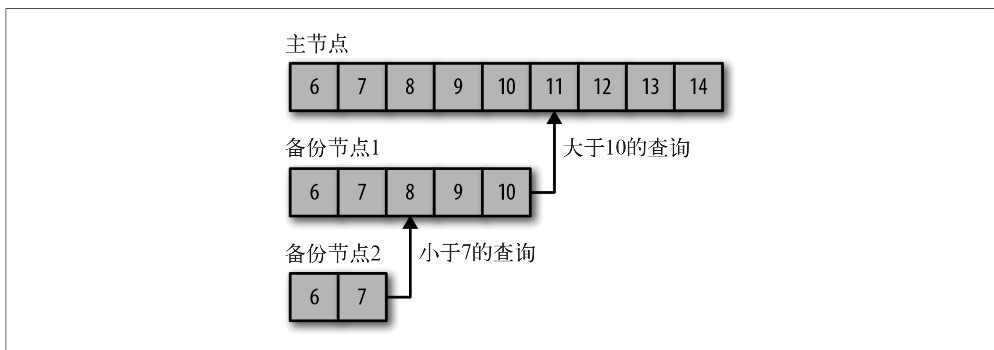


图 10-1：oplog 中按顺序保存着所有执行过的写操作。每个成员都维护着一份自己的 oplog，每个成员的 oplog 都应该跟主节点的 oplog 完全一致（可能会有一些延迟）

如果某个备份节点由于某些原因挂掉了，当它重新启动之后，就会自动从 oplog 中最后一个操作开始进行同步。由于复制操作的过程是先复制数据再写入 oplog，所以，备份节点可能会在已经同步过的数据上再次执行复制操作。MongoDB 在设计之初就考虑到了这种情况：将 oplog 中的同一个操作执行多次，与只执行一次的效果是一样的。

由于 oplog 大小是固定的，它只能保存特定数量的操作日志。通常，oplog 使用空间的增长速度与系统处理写请求的速率近乎相同：如果主节点上每分钟处理了 1 KB 的写入请求，那么 oplog 很可能也会在一分钟内写入 1 KB 条操作日志。但是，有一些例外情况：如果单次请求能够影响到多个文档（比如删除多个文档或者是多文档更新），oplog 中就会出现多条操作日志。如果单个操作会影响多个文档，那么每个受影响的文档都会对应 oplog 中的一条日志。因此，如果执行 `db.coll.remove()` 删除了 1 000 000 个文档，那么 oplog 中就会有 1 000 000 条操作日志，每条日志对应一个被删除的文档。如果执行大量的批量操作，oplog 很快就会被填满。

10.1.1　初始化同步

副本集中的成员启动之后，就会检查自身状态，确定是否可以从某个成员那里进行同步。如果不行的话，它会尝试从副本的另一个成员那里进行完整的数据复制。这个过程就是*初始化同步*（initial syncing），包括几个步骤，可以从 mongod 的日志中看到。

(1) 首先，这个成员会做一些记录前的准备工作：选择一个成员作为同步源，在 local.me 中为自己创建一个标识符，删除所有已存在的数据库，以一个全新的状态开始进行同步：

```
Mon Jan 30 11:09:18 [rsSync] replSet initial sync pending
Mon Jan 30 11:09:18 [rsSync] replSet syncing to: server-1:27017
Mon Jan 30 11:09:18 [rsSync] build index local.me { _id: 1 }
Mon Jan 30 11:09:18 [rsSync] build index done 0 records 0 secs
Mon Jan 30 11:09:18 [rsSync] replSet initial sync drop all databases
Mon Jan 30 11:09:18 [rsSync] dropAllDatabasesExceptLocal 1
```

注意，在这个过程中，所有现有的数据都会被删除。应该只在不需要保留现有数据的情况下做初始化同步（或者将数据移到其他地方），因为 mongod 会首先将现有数据删除。

(2) 然后是克隆（cloning），就是将同步源的所有记录全部复制到本地。这通常是整个过程中最耗时的部分：

```
Mon Jan 30 11:09:18 [rsSync] replSet initial sync clone all databases
Mon Jan 30 11:09:18 [rsSync] replSet initial sync cloning db: db1
Mon Jan 30 11:09:18 [fileAllocator] allocating new datafile /data/db/db1.ns,
    filling with zeroes...
```

(3) 然后就进入 oplog 同步的第一步，克隆过程中的所有操作都会被记录到 oplog 中。如果有文档在克隆过程中被移动了，就可能会被遗漏，导致没有被克隆，对于这样的文档，可能需要重新进行克隆：

```
Mon Jan 30 15:38:36 [rsSync] oplog sync 1 of 3
Mon Jan 30 15:38:36 [rsBackgroundSync] replSet syncing to: server-1:27017
Mon Jan 30 15:38:37 [rsSyncNotiǸer] replset setting oplog notiǸer to
    server-1:27017
Mon Jan 30 15:38:37 [repl writer worker 2] replication update of non-mod
    failed:
    { ts: Timestamp 1352215827000|17, h: -5618036261007523082, v: 2, op: "u",
      ns: "db1.someColl", o2: { _id: ObjectId('50992a2a7852201e750012b7') },
      o: { $set: { count.0: 2, count.1: 0 } } }
Mon Jan 30 15:38:37 [repl writer worker 2] replication info
    adding missing object
Mon Jan 30 15:38:37 [repl writer worker 2] replication missing object
    not found on source. presumably deleted later in oplog
```

上面是一个比较粗略的日志，显示了有文档需要重新克隆的情况。在克隆过程中也可能不会遗漏文档，这取决于流量等级和同步源上的操作类型。

(4) 接下来是 oplog 同步过程的第二步，用于将第一个 oplog 同步中的操作记录下来。

```
Mon Jan 30 15:39:41 [rsSync] oplog sync 2 of 3
```

这个过程比较简单，也没有太多的输出。只有在没有东西需要克隆时，这个过程才会与第一个不同。

(5) 到目前为止，本地的数据应该与主节点在某个时间点的数据集完全一致了，可

以开始创建索引了。如果集合比较大，或者要创建的索引比较多，这个过程会很耗时间：

```
Mon Jan 30 15:39:43 [rsSync] replSet initial sync building indexes
Mon Jan 30 15:39:43 [rsSync] replSet initial sync cloning indexes for : db1
Mon Jan 30 15:39:43 [rsSync] build index db.allObjects { someColl: 1 }
Mon Jan 30 15:39:44 [rsSync] build index done. scanned 209844 total records.
    1.96 secs
```

(6) 如果当前节点的数据仍然远远落后于同步源，那么 oplog 同步过程的最后一步就是将创建索引期间的所有操作全部同步过来，防止该成员成为备份节点。

```
Tue Nov 6 16:05:59 [rsSync] oplog sync 3 of 3
```

(7) 现在，当前成员已经完成了初始化同步，切换到普通同步状态，这时当前成员就可以成为备份节点了：

```
Mon Jan 30 16:07:52 [rsSync] replSet initial sync done
Mon Jan 30 16:07:52 [rsSync] replSet syncing to: server-1:27017
Mon Jan 30 16:07:52 [rsSync] replSet SECONDARY
```

如果想跟踪初始化同步过程，最好的方式就是查看服务器日志。

从操作者的角度来说，初始化同步是非常简单的：使用空的数据目录启动 mongod 即可。但是，更多时候可能需要从备份中恢复（第 22 章会详细介绍）而不是进行初始化同步。从备份中恢复的速度比使用 mongod 复制全部数据的速度快得多。

克隆也可能损坏同步源的工作集（working set）。实际部署之后，可能会有一个频繁使用的数据子集常驻内存（因为操作系统要频繁访问这个子集）。执行初始化同步时，会强制将当前成员的所有数据分页加载到内存中，这会导致需要频繁访问的数据不能常驻内存，所以会导致很多请求变慢，因为原本只要在 RAM（内存）中就可以处理的数据要先从磁盘上加载。不过，对于比较小的数据集和性能比较好的服务器，初始化同步仍然是个简单易用的选项。

初始化同步过程中经常遇到的问题是，第 (2) 步（克隆）或者第 (5) 步（创建索引）耗费了太长的时间。这种情况下，新成员就与同步源的 oplog"脱节"：新成员远远落后于同步源，导致新成员的数据同步速度赶不上同步源的变化速度，同步源可能会将新成员需要复制的某些数据覆盖掉。

这个问题没有有效的解决办法，除非在不太忙时执行初始化同步，或者是从备份中恢复数据。如果新成员与同步源的 oplog 脱节，初始化同步就无法正常进行。下一节会更深入地介绍。

10.1.2 处理陈旧数据

如果备份节点远远落后于同步源当前的操作，那么这个备份节点就是*陈旧的*（stale）。陈旧的备份节点无法跟上同步源的节奏，因为同步源上的操作领先太多太多：如果要继续进行同步，备份节点需要跳过一些操作。如果从备份节点曾经停机过，写入量超过了自身处理能力，或者是有太多的读请求，这些情况都可能导致备份节点陈旧。

当一个备份节点陈旧之后，它会查看副本集中的其他成员，如果某个成员的 oplog 足够详尽，可以用于处理那些落下的操作，就从这个成员处进行同步。如果任何一个成员的 oplog 都没有参考价值，那么这个成员上的复制操作就会中止，这个成员需要重新进行完全同步（或者是从最近的备份中恢复）。

为了避免陈旧备份节点的出现，让主节点使用比较大的 oplog 保存足够多的操作日志是很重要的。大的 oplog 会占用更多的磁盘空间。通常来说，这是一个比较好的折衷选择，因为磁盘会越来越便宜，而且实际中使用的 oplog 只有一小部分，因此 oplog 不占用太多 RAM。关于 oplog 空间占用的更多信息，12.4.6 节会详细介绍。

10.2 心跳

每个成员都需要知道其他成员的状态：哪个是主节点？哪个可以作为同步源？哪个挂掉了？为了维护集合的最新视图，每个成员每隔两秒钟就会向其他成员发送一个*心跳请求*（heartbeat request）。心跳请求的信息量非常小，用于检查每个成员的状态。

心跳最重要的功能之一就是让主节点知道自己是否满足集合“大多数”的条件。如果主节点不再得到“大多数”服务器的支持，它就会退位，变成备份节点。

成员状态

各个成员会通过心跳将自己的当前状态告诉其他成员。我们已经讨论过两种状态了：主节点和备份节点。还有其他一些常见状态。

- `STARTUP`
  成员刚启动时处于这个状态。在这个状态下，MongoDB 会尝试加载成员的副本集配置。配置加载成功之后，就进入 `STARTUP2` 状态。

- `STARTUP2`
  整个初始化同步过程都处于这个状态，但是如果是在普通成员上，这个状态只会

持续几秒钟。在这个状态下，MongoDB 会创建几个线程，用于处理复制和选举，然后就会切换到 RECOVERING 状态。

- RECOVERING

  这个状态表明成员运转正常，但是暂时还不能处理读取请求。如果有成员处于这个状态，可能会造成轻微的系统过载，以后可能会经常见到。

  启动时，成员需要做一些检查以确保自己处于有效状态，之后才可以处理读取请求。在启动过程中，成为备份节点之前，每个成员都要经历 RECOVERING 状态。在处理非常耗时的操作时，成员也可能进入 RECOVERING 状态。，比如压缩或者是响应 replSetMaintenance 命令（详见 12.3.3 节）。

  当一个成员与其他成员脱节时，也会进入 RECOVERING 状态。通常来说，这时这个成员处于无效状态，需要重新同步。但是，成员这时并没有进入错误状态，因为它期望发现一个拥有足够详尽 oplog 的成员，然后继续同步 oplog，最后回到正常状态。

- ARBITER

  在正常的操作中，仲裁者应该始终处于 ARBITER 状态。

系统出现问题时会处于下面这些状态。

- DOWN

  如果一个正常运行的成员变得不可达，它就处于 DOWN 状态。注意，如果有成员被报告为 DOWN 状态，它有可能仍然处于正常运行状态，不可达的原因可能是网络问题。

- UNKNOWN

  如果一个成员无法到达其他任何成员，其他成员就无法知道它处于什么状态，会将其报告为 UNKNOWN 状态。通常，这表明这个未知状态的成员挂掉了，或者是两个成员之间存在网络访问问题。

- REMOVED

  当成员被移出副本集时，它就处于这个状态。如果被移出的成员又被重新添加到副本集中，它就会回到“正常”状态。

- ROLLBACK

  如果成员正在进行数据回滚（详见 10.4 节），它就处于 ROLLBACK 状态。回滚过程结束时，服务器会转换为 RECOVERING 状态，然后成为备份节点。

- `FATAL`

  如果一个成员发生了不可挽回的错误，也不再尝试恢复正常的话，它就处于`FATAL`状态。应该查看详细日志来查明为何这个成员处于`FATAL`状态（使用`"replSet  FATAL"`关键词在日志上执行`grep`，就可以找到成员进入`FATAL`状态的时间点）。这时，通常应该重启服务器，进行重新同步或者是从备份中恢复。

10.3　选举

当一个成员无法到达主节点时，它就会申请被选举为主节点。希望被选举为主节点的成员，会向它能到达的所有成员发送通知。如果这个成员不符合候选人要求，其他成员可能会知道相关原因：这个成员的数据落后于副本集，或者是已经有一个运行中的主节点（那个力求被选举成为主节点的成员无法到达这个主节点）。在这些情况下，其他成员不会允许进行选举。

假如没有反对的理由，其他成员就会对这个成员进行选举投票。如果这个成员得到副本集中"大多数"赞成票，它就选举成功，会转换到主节点状态。如果达不到"大多数"的要求，那么选举失败，它仍然处于备份节点状态，之后还可以再次申请被选举为主节点。主节点会一直处于主节点状态，除非它由于不再满足"大多数"的要求或者挂了而退位，另外，副本集被重新配置也会导致主节点退位。

假如网络状况良好，"大多数"服务器也都在正常运行，那么选举过程是很快的。如果主节点不可用，2 秒钟（之前讲过，心跳的间隔是 2 秒）之内就会有成员发现这个问题，然后会立即开始选举，整个选举过程只会花费几毫秒。但是，实际情况可能不会这么理想：网络问题，或者是服务器过载导致响应缓慢，都可能触发选举。在这种情况下，心跳会在最多 20 秒之后超时。如果选举打成平局，每个成员都需要等待 30 秒才能开始下一次选举。所以，如果有太多错误发生的话，选举可能会花费几分钟的时间。

10.4　回滚

根据上一节讲述的选举过程，如果主节点执行了一个写请求之后挂了，但是备份节点还没来得及复制这次操作，那么新选举出来的主节点就会漏掉这次写操作。假如有两个数据中心，其中一个数据中心拥有一个主节点和一个备份节点，另一个数据中心拥有三个备份节点，如图 10-2 所示。

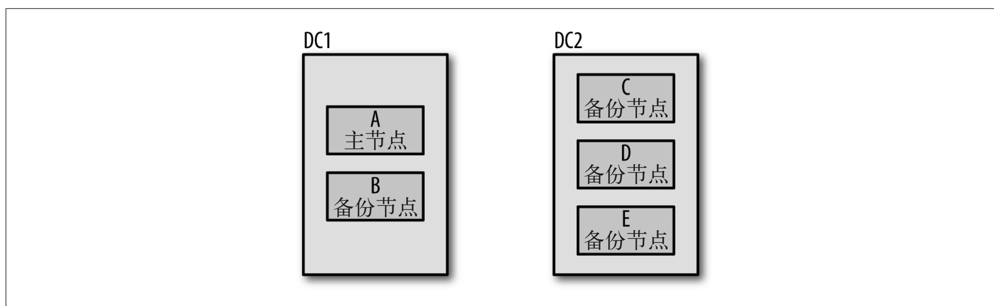


图 10-2：一个可能的双数据中心配置

如果这两个数据中心之间出现了网络故障，如图 10-3 所示。其中左边第一个数据中心最后的操作是 126，但是 126 操作还没有被复制到另边的数据中心。

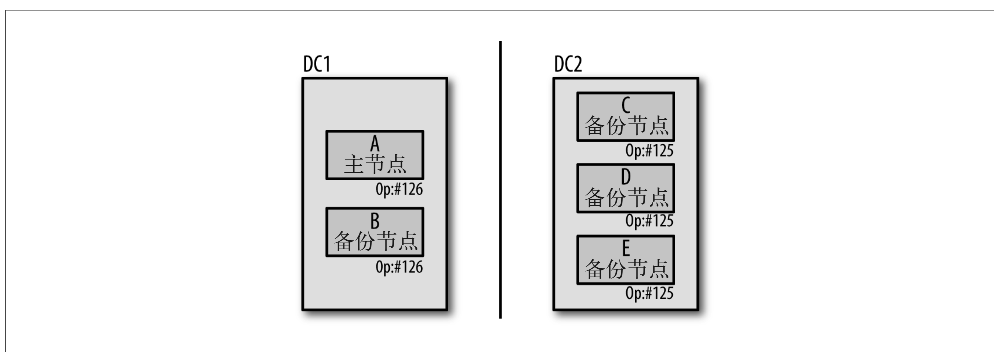


图 10-3：在不同数据中心之间进行复制比在单一数据中心内要慢

右边的数据中心仍然满足副本集“大多数”的要求（一共 5 台服务器，3 台即可满足要求）。因此，其中一台服务器会被选举成为新的主节点，这个新的主节点会继续处理后续的写入操作，如图 10-4 所示。

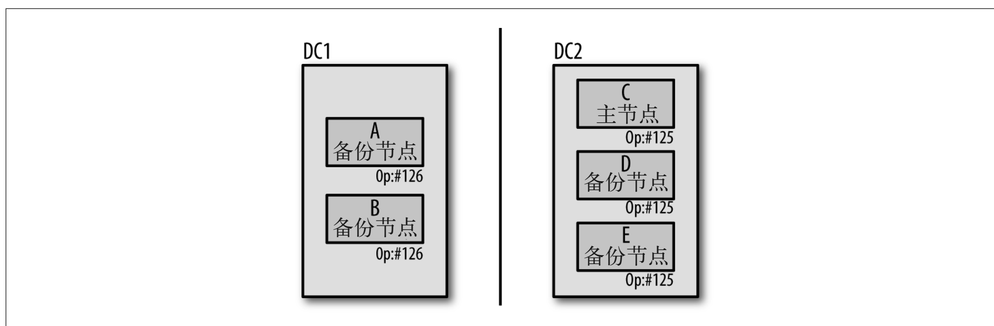


图 10-4：右边数据中心未能完成复制左边数据中心的写操作

网络恢复之后，左边数据中心的服务器就会从其他服务器开始同步 126 之后的操作，但是无法找到这个操作。这种情况发生的时候，A 和 B 会进入回滚（rollback）过程。回滚会将失败之前未复制的操作撤消。拥有 126 操作的服务器会在右边数据中心服务器的 oplog 中寻找共同的操作点。之后会定位到 125 操作，这是两个数据中心相匹配的最后一个操作。图 10-5 显示了 oplog 的情况。

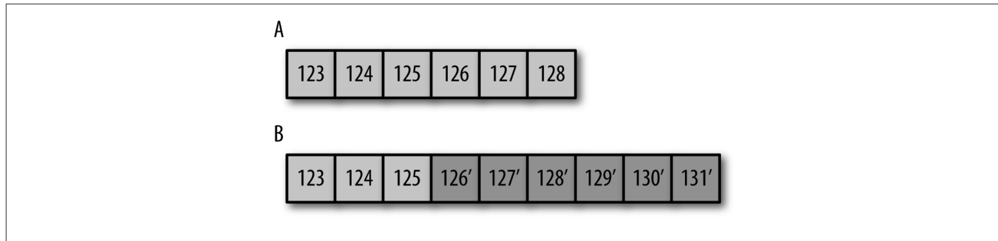


图 10-5：图中两个成员的 oplog 有冲突：很显然，A 的 126-128 操作被复制之前，A 崩溃了，所以这些操作并没有出现在 B 中（B 拥有更多的最近操作）。A 必须先将 126-128 这 3 个操作回滚，然后才能重新进行同步

这时，服务器会查看这些没有被复制的操作，将受这些操作影响的文档写入一个 .bson 文件，保存在数据目录下的 rollback 目录中。如果 126 是一个更新操作，服务器会将被 126 更新的文档写入 collectionName.bson 文件。然后会从当前主节点中复制这个文档。

下面是一次典型的回滚过程产生的日志：

```
Fri Oct  7 06:30:35 [rsSync] replSet syncing to: server-1
Fri Oct  7 06:30:35 [rsSync] replSet our last op time written: Oct 7
    06:30:05:3
Fri Oct  7 06:30:35 [rsSync] replset source's GTE: Oct 7 06:30:31:1
Fri Oct  7 06:30:35 [rsSync] replSet rollback 0
Fri Oct  7 06:30:35 [rsSync] replSet ROLLBACK
Fri Oct  7 06:30:35 [rsSync] replSet rollback 1
Fri Oct  7 06:30:35 [rsSync] replSet rollback 2 ÑndCommonPoint
Fri Oct  7 06:30:35 [rsSync] replSet info rollback our last optime: Oct 7
    06:30:05:3
Fri Oct  7 06:30:35 [rsSync] replSet info rollback their last optime: Oct 7
    06:30:31:2
Fri Oct  7 06:30:35 [rsSync] replSet info rollback diff in end of log times:
    -26 seconds
Fri Oct  7 06:30:35 [rsSync] replSet rollback found matching events at Oct 7
    06:30:03:4118
Fri Oct  7 06:30:35 [rsSync] replSet rollback Ñndcommonpoint scanned : 6
Fri Oct  7 06:30:35 [rsSync] replSet replSet rollback 3 Ñxup
Fri Oct  7 06:30:35 [rsSync] replSet rollback 3.5
Fri Oct  7 06:30:35 [rsSync] replSet rollback 4 n:3
Fri Oct  7 06:30:35 [rsSync] replSet minvalid=Oct 7 06:30:31 4e8ed4c7:2
```

```
Fri Oct  7 06:30:35 [rsSync] replSet rollback 4.6
Fri Oct  7 06:30:35 [rsSync] replSet rollback 4.7
Fri Oct  7 06:30:35 [rsSync] replSet rollback 5 d:6 u:0
Fri Oct  7 06:30:35 [rsSync] replSet rollback 6
Fri Oct  7 06:30:35 [rsSync] replSet rollback 7
Fri Oct  7 06:30:35 [rsSync] replSet rollback done
Fri Oct  7 06:30:35 [rsSync] replSet RECOVERING
Fri Oct  7 06:30:36 [rsSync] replSet syncing to: server-1
Fri Oct  7 06:30:36 [rsSync] replSet SECONDARY
```

服务器开始从另一个成员进行同步（在本例中是 server-1），但是发现无法在同步源中找到自己的最后一次操作。这时，它就会切换到回滚状态（"replSet ROLLBACK"）进行回滚。

第 2 步，服务器在两个 oplog 中找到一个共同的点，是 26 秒之前的一个操作。然后服务器就会将最近 26 秒内执行的操作从 oplog 中撤销。回滚完成之后，服务器就进入 RECOVERING 状态开始进行正常同步。

如果要将被回滚的操作应用到当前主节点，首先使用 mongorestore 命令将它们加载到一个临时集合：

```
$ mongorestore --db stage --collection stuff \
> /data/db/rollback/important.stuff.2012-12-19T18-27-14.0.bson
```

现在应该在 shell 中将这些文档与同步后的集合进行比较。例如，如果有人在被回滚的成员上创建了一个“普通”索引，而当前主节点创建了一个唯一索引，那么就需要确保被回滚的数据中没有重复文档，如果有的话要去除重复。

如果希望保留 staging 集合中当前版本的文档，可以将其载入主集合：

```
> staging.stuff.find().forEach(function(doc) {
...     prod.stuff.insert(doc);
... })
```

对于只允许插入的集合，可以直接将被回滚的文档插入主集合。但是，如果是在集合上执行更新操作，在合并回滚数据时就要非常小心地对待。

一个经常会被误用的成员配置选项是设置每个成员的投票数量。改变成员的投票数量通常不会得到想要的结果，而且很可能会导致大量的回滚操作（所以上一章的成员属性列表中没有介绍这个选项）。除非做好了定期处理回滚的准备，否则不要改变成员的投票数量。

第 11 章会讲述如何阻止回滚。

如果回滚失败

某些情况下，如果要回滚的内容太多，MongoDB 可能承受不了。如果要回滚的数据量大于 300 MB，或者要回滚 30 分钟以上的操作，回滚就会失败。对于回滚失败的节点，必须要重新同步。

这种情况最常见的原因是备份节点远远落后于主节点，而这时主节点却挂了。如果其中一个备份节点成为主节点，这个主节点与旧的主节点相比，缺少很多操作。为了保证成员不会在回滚中失败，最好的方式是保持备份节点的数据尽可能最新。

第 11 章

从应用程序连接副本集

本章介绍如何在应用程序中与副本集进行交互，包括：

- 如何连接到副本集以及故障转移的工作机制；
- 等待写入复制；
- 将读请求路由到正确的成员。

11.1　客户端到副本集的连接

从应用程序的角度来说，使用副本集与使用单台服务器很像。默认情况下，驱动程序会连接到主节点，并且将所有请求都路由到主节点。应用程序可以像使用单台服务器一样进行读和写，副本集会在后台默默处理热备份。

连接副本集与连接单台服务器非常像。在驱动程序中使用与 `MongoClient` 等价的对象，并且提供一个希望连接到的副本集种子（seed）列表。种子是副本集成员，并不需要将所有成员都列出来（虽然可以这么做）：驱动程序连接到某个种子服务器之后，就能够得到其他成员的地址。一个常用的连接字符串如下所示：

```
"mongodb://server-1:27017,server-2:27017"
```

具体可以查看相关的驱动程序文档。

当主节点挂掉之后，驱动程序会尽快自动找到新的主节点（只要新的主节点被选举出来），并且将请求路由到新的主节点。但是，如果没有可达的主节点，应用程序就

无法执行写操作。

在选举过程中，主节点可能会暂时不可用；如果没有可达的成员能够成为主节点，主节点可能长时间不可用。默认情况下，驱动程序在这段时间内不会处理任何请求（读或写）。但是，可以选择将读请求路由到备份节点。

从用户的角度来说，希望驱动程序能够隐藏掉整个选举过程（主节点退位，新的主节点被选举出来）。但是，在很多情况下这是不可能做到的，所以没有哪个驱动程序能够这样处理故障转移。首先，驱动程序仅仅能够将没有主节点的情况隐瞒一段时间：副本集不能在没有主节点的情况下永久存在。其次，如果有操作失败了，驱动程序就知道是主节点挂了，但是无法知道主节点在挂掉之前是否已经正确处理本次请求。所以，驱动程序将这个问题留给了用户：如果新的主节点很快被选举出来，要不要在新的主节点上重新操作？是否要假设最后一次请求已经被旧的主节点处理完成？是否要检查新的主节点以确保它同步了最后的操作？对这些具体问题的处理都取决于你的应用程序。

通常，驱动程序没有办法判断某次操作是否在服务器崩溃之前成功处理，但是应用程序可以自己实现相应的解决方案。比如，如果驱动程序发出插入 `{"_id" : 1}` 文档的请求之后收到主节点崩溃的错误，连接到新的主节点之后，可以查询主节点中是否有 `{"_id" : 1}` 这个文档。

11.2 等待写入复制

前面章节中已经提到，如果希望不管发生什么都将写入操作保存到副本集中，那么必须要确保写入操作被同步到了副本集的“大多数”。

之前，我们使用 `getLastError` 命令检查写入是否成功。也可以使用这个命令确保写入操作被复制到备份节点。参数 `"w"` 会强制要求 `getLastError` 等待，一直到给定数量的成员都执行完了最后的写入操作。MongoDB 有一个特殊的关键字可以传递给 `"w"`，就是 `"majority"`。在 shell 中它如下所示：

```
> db.runCommand({"getLastError" : 1, "w" : "majority"})
{
    "n" : 0,
    "lastOp" : Timestamp(1346790783000, 1),
    "connectionId" : 2,
    "writtenTo" : [
        { "_id" : 0 , "host" : "server-0" },
        { "_id" : 1 , "host" : "server-1" },
        { "_id" : 3 , "host" : "server-3" }
    ],
    "wtime" : 76,
```

```
    "err" : null,
    "ok" : 1
}
```

注意，getLastError 输出信息中的新字段 "writtenTo"。只有当使用了 "w" 选项并且最后的操作被复制到多个服务器时才会有这个字段。

假设在执行这个命令时只有主节点和一个仲裁者节点可用，那么主节点就无法将这个写操作复制到副本集中的任何成员。getLastError 并不知道应该等待多久，所以它会一直等待下去。因此，应该始终为 wtimeout 选项设置一个合理的值。"wtimeout" 是 getLastError 可以使用的另一个选项，它的值是命令的超时时间，如果超过这个时间还没有返回，就会返回失败：MongoDB 无法在指定时间内将写入操作复制到 "w" 个成员。

下列代码的超时时间是 1 秒钟：

```
> db.runCommand({"getLastError" : 1, "w" : "majority", "wtimeout" : 1000})
```

这个命令可能会由于多种原因失败：其他成员可能挂了，可能落后于主节点，也可能由于网络问题不可访问。如果 getLastError 超时，应用程序必须要对这种情况作出处理。注意，getLastError 超时并不意味着写操作失败了，仅仅表明写操作没能在指定时间内复制到足够多的成员。写操作仍然被复制到了一些成员，而且会尽快传播到其他成员。

通常将 "w" 用于控制写入速度。MongoDB 的写入速度“太快”，主节点上执行完写入操作之后，备份节点还来不及跟上。阻止这种行为的一种常用方式是定期调用 getLastError，将 "w" 参数指定为大于 1 的值。这样就会强制这个连接上的写操作一直等待直到复制成功。注意，这只会阻塞这个连接上的写操作：其他连接上的写操作仍然会立即执行完成并返回。

如果希望应用程序的行为更自然更健壮，应该定期调用 getLastError，同时指定 "majority" 和一个合理的超时时间。如果这个命令超时了，需要找出出错原因。

11.2.1　可能导致错误的原因

假设应用程序将一个写操作发送给主节点，然后调用 getLastError（不使用 "majority" 选项）收到写入成功的反馈，但是在备份节点复制这个写操作之前，主节点崩溃了。

现在，应用程序认为可以访问之前的写操作（getLastError 命令的输出信息表明写入操作成功完成），但是副本集中的当前成员并不拥有这个操作的副本。

在某个时刻，会有一个备份节点被选举为新的主节点，然后开始接受新的写请求。当之前的主节点恢复之后，会发现它拥有一个（或几个）主节点上没有的写操作。为了纠正这个问题，它会撤销与当前主节点不一致的操作。这些操作不会丢失，但是会被写到特殊的回滚文件中，之后可以手动将这些操作应用到当前主节点。MongoDB 不能自动应用这些写操作，因为这些写操作可能会与崩溃之后产生的其他操作冲突。因此，这些操作会消失，直到管理员将这些操作应用到当前主节点。

写入时指定 `majority` 可以避免这种情况的发生：如果应用程序最初使用 `"w" : "majority"` 并且得到了写入成功的确认信息，那么新的主节点就拥有之前执行过的写操作（一个成员必须足够新，才能被选举为主节点）。如果 `getLastError` 失败，应用程序就会知道在操作被复制到大多数成员之前主节点就挂了，应用程序可以重新执行这个操作。

关于回滚的详细信息可以查看第 10 章。

11.2.2 "w"的其他值

`"majority"` 并不是唯一一个可以传递给 `getLastError` 的 `"w"` 参数的值，MongoDB 允许将 `"w"` 指定为任意整数，如下所示：

```
> db.runCommand({"getLastError" : 1, "w" : 2, "wtimeout" : 500})
```

这个命令会一直等待，直到写操作被复制到两个成员（主节点和一个备份节点）。

注意，`"w"` 的值包含了主节点。如果希望写操作被复制到 `n` 个备份节点，应该将 `"w"` 指定为 `n+1`（包括主节点）。将 `"w"` 设置为 1 相当于没有传入 `"w"` 选项，因为 MongoDB 只会检查主节点是否成功执行了写操作，`getLastError` 始终会做这样的检查。

使用常量数值的弊端在于，如果副本集的配置发生了变化，就需要修改你的应用程序。

11.3 自定义复制保证规则

写入副本集的“大多数”成员被认为是安全写入。然而，有些副本集可能有更复杂的要求：可能会希望确保写操作被复制到每个数据中心中至少一台服务器上，或者是被复制到可见节点的“大多数”服务器上。副本集允许创建自己的规则，并且可以传递给 `getLastError`，以保证写操作被复制到所需的服务器上。

11.3.1 保证复制到每个数据中心的一台服务器上

相对于单个数据中心内部，不同数据中心之间更容易发生网络故障；相对于多个数

据中心同等数量的服务器挂掉，整个数据中心挂掉的可能性更高。因此，可能你希望有一些针对数据中心的逻辑来保证写操作成功执行。在确认成功之前，保证写操作被复制到每一个数据中心，这样，万一某个数据中心掉线了，其他每一个数据中心都有一份最新的本地数据副本。

要实现这种机制，首先按照数据中心对成员分类。可以在副本集配置中添加一个 `"tags"` 字段：

```
> var config = rs.config()
> config.members[0].tags = {"dc" : "us-east"}
> config.members[1].tags = {"dc" : "us-east"}
> config.members[2].tags = {"dc" : "us-east"}
> config.members[3].tags = {"dc" : "us-east"}
> config.members[4].tags = {"dc" : "us-west"}
> config.members[5].tags = {"dc" : "us-west"}
> config.members[6].tags = {"dc" : "us-west"}
```

`"tags"` 字段是一个对象，每个成员可以拥有多个标签。例如，`"us-east"` 数据中心的服务器可能是 `"high quality"` 服务器，这样的话，可以将其 `"tags"` 字段配置为 `{"dc": "us-east", "quality" : "high"}`。

第二步是创建自己的规则，可以通过在副本集配置中创建 `"getLastErrorMode"` 字段实现。每条规则的形式都是 `"`*`name`*`" : {"`*`key`*`" : `*`number`*`}}`。`"name"` 就是规则的名称，名称应该能够表明这条规则所做的事情，方便客户端理解，客户端在调用 `getLastError` 时才能够正确选择自己需要的规则。在本例中，将这个规则命名为 `"eachDC"`，或者更抽象一点，比如 `"user-level safe"`。

这里的 `"`*`key`*`"` 字段就是标签键的值，所以在这个例子中是 `"dc"`。这里的 *`number`* 是需要遵循这条规则的分组的数量。在本例中，*`number`* 是 2（因为我们希望写操作被复制到 `"us-east"` 和 `"us-west"` 两个分组中各自至少一台服务器）。*`number`* 的意思是“保证写操作复制到 *`number`* 个分组，每个分组内至少一台服务器上”。

在副本集配置中添加 `"getLastErrorModes"` 字段，创建下面的规则，重新执行配置：

```
> config.settings = {}
> config.settings.getLastErrorModes = [{"eachDC" : {"dc" : 2}}]
> rs.reconfig(config)
```

`"getLastErrorModes"` 位于副本集配置中的 `"settings"` 子字段，这个字段下面包含一些副本集级别的可选设置。

现在，可以对写操作应用这条规则：

```
> db.foo.insert({"x" : 1})
> db.runCommand({"getLastError" : 1, "w" : "eachDC", "wtimeout" : 1000})
```

注意，应用程序开发者并不会知道到底有哪些服务器使用了 "eachDC" 规则，而且可以在不改变应用程序的情况下任意修改具体规则。可以添加新的数据中心，或者是更改副本集成员数量，而应用程序不必知道这些改变。

11.3.2　保证写操作被复制到可见节点中的“大多数”

通常，隐藏节点在某种程度上是二等公民：发生故障时不会转移到隐藏节点，也不能将读操作路由到隐藏节点。你可能只关心隐藏节点是否收到了写请求，剩下的就交给隐藏成员自己去解决吧。

假设我们拥有 5 个成员，host0 到 host4，其中 host4 是个隐藏成员。我们希望确保写操作被复制到非隐藏节点的大多数，也就是 host0、host1、host2 和 host3 中的至少三个成员。要创建这样一条规则，首先为非隐藏节点设置标签：

```
> var config = rs.config()
> config.members[0].tags = [{"normal" : "A"}]
> config.members[1].tags = [{"normal" : "B"}]
> config.members[2].tags = [{"normal" : "C"}]
> config.members[3].tags = [{"normal" : "D"}]
```

不需要为隐藏节点（host4）设置标签。

现在，为这些服务器中的大多数添加这条规则：

```
> conÑg.settings.getLastErrorModes = [{"visibleMajority" : {"normal" : 3}}]
> rs.reconÑg(conÑg)
```

然后就可以在应用程序中使用这条规则了：

```
> db.foo.insert({"x" : 1})
> db.runCommand({"getLastError" : 1, "w" : "visibleMajority", "wtimeout": 1000})
```

命令会一直等待，直到写操作被复制到至少三个非隐藏节点。

11.3.3　创建其他规则

可以无限制地创建各种规则。记住，创建自定义的复制规则有两个步骤。

(1) 使用键值对设置成员的 "tags" 字段。这里的键用于描述分组，可能会有 "data_center"、"region" 或者 "server Quality" 等键。这里的值表示服务器所属的分组。例如，对于“data_center”这个键，可以将一些服务器标为 "us-east"，将另一些标为 "us-west"，其他的标为 "aust"。

(2) 基于刚刚创建的分组创建规则。规则总是形如 {"*name*" : {"*key*" :

number}}，表示写操作返回成功之前需要复制到至少 *number* 个分组，每个分组内的一台服务器上。例如，可以创建一个 {"twoDCs" : {"data_center" : 2}} 规则，意思是说，在写操作成功之前，需要确保写操作被复制到两个数据中心，每个数据中心内至少一台服务器上。

然后就可以在 getLastError 中使用刚刚创建的规则了。

规则是一种非常强大的副本集配置方式，虽然它理解和设置起来都有些复杂。除非有非常特殊的复制要求，否则使用 "w":"majority" 就已经非常安全了。

11.4 将读请求发送到备份节点

默认情况下，驱动程序会将所有的请求都路由到主节点。这通常也正是你需要的，但是可以通过设置驱动程序的*读取首选项*（read preferences）配置其他选项。可以在读选项中设置需要将查询路由到的服务器的类型。

将读请求发送到备份节点通常不是一个好主意。虽然在某些特定情况下这是有意义的，但是通常应该将全部请求都路由到主节点。如果你正在考虑将读请求发送到备份节点，请先从各个方面好好权衡之后再做决定。本节会说明不建议这么做的原因，也会介绍需要这么做的特定情况。

11.4.1 出于一致性考虑

对一致性要求非常高的应用程序不应该从备份节点读取数据。

备份节点通常会落后主节点几毫秒，但是，不能保证一定是这样。有时，由于加载问题、配置错误、网络故障等原因，备份节点可能会落后于主节点几分钟、几个小时甚至几天。客户端驱动程序并不知道备份节点的数据有多新，所以如果将读请求发送给一个远远落后于主节点的备份节点，客户端也不会感觉到任何问题。可以将备份节点隐藏掉，以避免客户端读取它，但是这是一个手动过程。如果你的应用程序需要读取最新的数据，那就不要从备份节点读取数据。

如果应用程序需要读取它自己的写操作（例如，先插入一个文档，然后再查询它），那么不应该将写请求发送给备份节点（除非写操作像前面那样，使用 "w" 在返回之前被复制到所有备份节点）。否则的话，可能会出现应用程序成功执行了一次写操作，却读不到这个值的情况（因为读请求被发送给了备份节点，而之前的写操作还没有被复制到这个备份节点）。客户端发送请求的速度可能会比备份节点复制操作的速度要快。

为了能够始终将写请求发送给主节点，需要将读选项设置为 `Primary`（或者不管它，默认就是 `Primary`）。如果没有主节点，查询就会出错。这就是说，如果主节点挂了，应用程序就不能执行查询了。但是，如果你的应用程序需要在故障转移期间或者出现网络故障时正常运行，或者不接受陈旧的数据，那么这就是一个可接受的选项。

11.4.2　出于负载的考虑

许多用户会将读请求发送给备份节点，以便实现分布式负载。例如，如果你的服务器每秒只能处理 10 000 次查询，而你需要进行 30 000 次查询，可能就需要设置几个备份节点，并且让它们分担一些数据加载的工作。但是，这种扩展方式非常危险，很容易导致系统意外过载，一旦出现这种问题，很难恢复。

假设你遇到了上面提到的情况：每秒 30 000 次读请求。你决定创建一个拥有 4 个成员的副本集：每个备份节点的压力都在可承受范围内，整个系统也在正常运转。

后来，某一个备份节点崩溃了。

现在剩余的每个成员的负载都是 100%。如果需要恢复刚刚崩溃的成员，它就需要从其他成员处复制数据，这就会导致其他成员过载。服务器过载经常导致性能变慢，副本集性能进一步降低，然后强制其他成员承担更多的负载，导致这些成员变得更慢，这是一个恶性死循环。

过载会导致副本集性能降低，然后会导致剩余的备份节点远远落后于主节点。突然间，你的副本集中就有一个成员崩溃了，还有一个成员远远落后于主节点，导致副本集的所有成员都过载了，进而整个副本集都没有喘息的空间。

如果明确知道每台服务器能够承受的负载，你可能会觉得自己能够更好地应对这种情况：使用 5 台服务器，而不是 4 台，这样如果一台服务器崩溃，并不会导致副本集过载。但是，即使你的计划非常完美（只有预期数量的服务器可能会挂掉），仍然需要处理其他服务器负载过大的情况。

一个更好的选择是，使用分片作分布式负载。第 13 章介绍分片相关的知识。

11.4.3　何时可以从备份节点读取数据

在某些情况下，将读请求发送给备份节点是合理的。例如，你可能希望应用程序在主节点挂掉时仍然能够执行读操作（而且你并不在意读到的数据是否是最新的）。这是最常见的将读请求发给送备份节点的原因：失去主节的时，应用程序进入只读状

态。这种读选项叫做*主节点优先*（primary preferred）。

从备份节点读取数据有一个常见的参数是获得低延迟的数据。可以将读选项设置为`Nearest`，以便将请求路由到延迟最低的成员（根据驱动程序到副本集成员的ping时间）。如果你的应用程序需要从多个数据中心中读取到最低延迟的同一个文档，这是唯一的方法。如果你的文档与位置的相关性更大（在这个数据中心内的应用服务器需要得到某些文档的最低延迟版本，而另一个数据中心内的应用服务器需要得到另一些文档的最低延迟版本），那就应该使用分片。注意，如果应用程序要求低延迟的读和写，那必须要使用分片：副本集只允许在主节点上进行写操作（不管主节点在什么位置）。

如果要从一个落后的备份节点读取数据，就要牺牲一致性。另一方面，如果希望写操作返回之前被复制到所有副本集成员，就要牺牲写入速度。

如果应用程序能够接受任何陈旧程序的数据，那就可以使用`Secondary`或者`Secondary preferred`读选项。`Secondary`始终会将读请求发送给备份节点。如果没有可用的备份节点，请求就会出错，而不是重新将读请求发送给主节点。对于不在乎数据新旧程度并且希望主节点只处理写请求的应用程序来说，这是一种可行的方式。如果对于数据新旧程度有要求，不建议使用这种方式。

`Secondary preferred`会优先将读请求路由到可用的备份节点。如果备份节点都不可用，请求就会被发送到主节点。

有时，读负载与写负载完全不同：读到的数据与写入的数据是完全不同的。为了做离线处理，你可能希望创建很多索引，但是又不想将这些索引创建在主节点上。在这种情况下，可以设置一个与主节点拥有不同索引的备份节点。如果希望以这种方式使用备份节点，最好是使用驱动程序创建一个直接连接到目标备份节点的连接，而不是连接到副本集。

应该根据应用程序的实际需要选择合适的选项。也可以将多个选项组合在一起使用：如果某些读请求必须从主节点读取数据，那就对这些请求使用`Primary`选项。如果另一些读请求并不要求数据是最新的，那么可以对这些读请求使用`Primary preferred`选项。如果某些请求对低迟延的要求大过一致性要求，那么可以使用`Nearest`选项。

第 12 章 管理

本章介绍副本集管理的相关知识，包括：

- 维护独立的成员；
- 在多种不同情况下配置副本集；
- 获取 oplog 相关信息，以及调整 oplog 大小；
- 特殊的副本集配置；
- 从主从模式切换到副本集模式 。

12.1 以单机模式启动成员

许多维护工作不能在备份节点上进行（因为要执行写操作），也不能在主节点上进行。后面几节会经常提到以单机模式（standalone mode）启动服务器。这是指要重启成员服务器，让它成为一个单机运行的服务器，而不再是一个副本集成员（这只是临时的）。

在以单机模式启动服务器之前，先看一下服务器的命令行参数：

```
> db.serverCmdLineOpts()
{
    "argv" : [ "mongod", "-f", "/var/lib/mongod.conf" ],
    "parsed" : {
        "replSet": "mySet",
        "port": "27017",
        "dbpath": "/var/lib/db"
    },
```

```
    "ok" : 1
}
```

如果要对这台服务器进行维护，可以重启服务器，重启时不使用 replSet 选项。这样它就会成为一个单机的 mongod，可以对其进行读和写。我们不希望副本集中的其他服务器联系到这台服务器，所以可以让它监听不同的端口（这样副本集的其他成员就找不到它了）。最后，保持 dbpath 的值不变，因为重启后要对这台服务器的数据做一些操作。好了，我们最终可以用下面这样的参数启动服务器：

```
$ mongod --port 30000 --dbpath /var/lib/db
```

现在这台服务器已经在单机模式中运行了，监听着 30000 端口的连接请求。副本集中的其他成员仍然会试图连接到它的 27017 端口，所以会连接失败，其他成员就会以为这台服务器挂掉了。

当在这台服务器上执行完维护工作之后，可以以最原始的参数重新启动它。启动之后，它会自动与副本集中的其他成员进行同步，将维护期间落下的操作全部复制过来。

12.2 副本集配置

副本集配置总是以一个文档的形式保存在 local.system.replSet 集合中。副本集中所有成员的这个文档都是相同的。绝对不要使用 update 更新这个文档，应该使用 rs 辅助函数或者 replSetReconfig 命令修改副本集配置。

12.2.1 创建副本集

创建副本集的步骤很简单，首先启动所有成员服务器，然后使用 rs.initiate 命令将配置文件传递给其中一个成员：

```
> var config = {
... "_id" : setName,
... "members" : [
...     {"_id" : 0, "host" : host1},
...     {"_id" : 1, "host" : host2},
...     {"_id" : 2, "host" : host3}
... ]}
> rs.initiate(config)
```

应该总是传递一个配置对象给 rs.initiate，否则 MongoDB 会自动生成一个针对单成员副本集的配置，其中的主机名可能不是你希望的。

只需要对副本集中的一个成员调用 rs.initiate 就可以了。收到 initiate 命令的成员会自动将配置文件传递给副本集中的其他成员。

12.2.2 修改副本集成员

向副本集中添加新成员时，这个新成员的数据目录要么是空的（在这种情况下，新成员会执行初始化同步），要么新成员拥有一份其他成员的数据副本。关于副本集成员备份和恢复相关的知识，可以查看第 22 章。

连接到主节点并且添加新成员：

```
> rs.add("spock:27017")
```

也可以以文档的形式为新成员指定更复杂的配置：

```
> rs.add({"_id" : 5, "host" : "spock:27017", "priority" : 0, "hidden" : true})
```

可以根据 "host" 字段将成员从副本集中移除：

```
> rs.remove("spock:27017")
```

可以通过 rs.reconfig 修改副本集成员的配置。修改副本集成员配置时，有几个限制需要注意：

- 不能修改成员的 "_id" 字段；
- 不能将接收 rs.reconfig 命令的成员（通常是主节点）的优先级设为 0；
- 不能将仲裁者成员变为非仲裁者成员，反之亦然；
- 不能将 "buildIndexes" : false 的成员修改为 "buildIndexes" : true。

需要注意的是，可以修改成员的 "host" 字段。这意味着，如果为副本集成员指定了不正确的主机名（比如使用了公网 IP 而不是内网 IP），之后可以重新修改成员的主机名。

下面是一个修改主机名的例子：

```
> var config = rs.config()
> config.members[0].host = "spock:27017"
spock:27017
> rs.reconfig(config)
```

修改其他选项的方式也是一样的：使用 rs.config 得到当前配置文件，修改配置文件，将修改后的配置文件传递给 rs.reconfig 就可以了。

12.2.3 创建比较大的副本集

副本集最多只能拥有 12 个成员，其中只有 7 个成员拥有投票权。这是为了减少心跳请求的网络流量（每个成员都要向其他所有成员发送心跳请求）和选举花费的时间。

实际上，副本集还有更多的限制，如果需要 11 个以上的备份节点，可以查看 12.5 节。

如果要创建 7 个以上成员的副本集，只有 7 个成员可以拥有投票权，需要将其他成员的投票数量设置为 0：

```
> rs.add({"_id" : 7, "host" : "server-7:27017", "votes" : 0})
```

这样可以阻止这些成员在选举中投主动票，虽然它们仍然可以投否决票。

应该尽量避免修改成员的投票数量。投票可能会对选举和一致性产生怪异的、不直观的影响。应该只在创建包含 7 个以上成员的副本集或者是希望阻止自动故障转移（详见 12.5.2 节）时，使用 "votes" 选项。很多开发者会误以为让成员拥有更多投票权会使这个成员更容易被选为主节点（实际上根本不会）。如果希望某个成员可以优先被选举为主节点，应该使用优先级（详见 9.6.2 节）。

12.2.4　强制重新配置

如果副本集无法再达到“大多数”要求的话，那么它就无法选举出新的主节点，这时你可能会希望重新配置副本集。这看起来有点奇怪，因为通常都是将配置文件发送给主节点。在这种情况下，可以在备份节点上调用 rs.reconfig 强制重新配置（force reconfigure）副本集。在 shell 中连接到一个备份节点，使用 "force" 选项执行 rs.reconfig 命令：

```
> rs.reconfig(config, {"force" : true})
```

强制重新配置与普通的重新配置要遵守同样的规则：必须使用正确的 reconfig 选项将有效的、格式完好的配置文件发送给成员。"force" 选项不允许无效的配置，而且只允许将配置发送给备份节点。

强制重新配置会跳过大量的数值直接将副本集的 "version" 设为一个比较大的值。可能会见到跳过数千的情况，这很正常：这是为了防止 "version" 字段冲突（以防不同的网络域中都在进行重新配置）。

备份节点收到新的配置文件之后，就会修改自身的配置，并且将新的配置发送给副本集中的其他成员。副本集的其他成员收到新的配置文件之后，会判断配置文件的发送者是否是它们当前配置中的一个成员，如果是，才会用新的配置文件对自己进行重新配置。所以，如果新的配置会修改某些成员的主机名，应该将新的配置发送给主机名不发生变化的成员。如果新的配置文件修改了所有成员的主机名，应该关闭副本集的每一个成员，以单机模式启动，手动修改 local.system.replset 文档，然后重新启动。

12.3 修改成员状态

为进行维护或响应加载，有多种方式可以手动修改成员的状态。注意，无法强制将某个成员变成主节点，除非对副本集做适当的配置。

12.3.1 把主节点变为备份节点

可以使用 stepDown 函数将主节点降级为备份节点：

```
> rs.stepDown()
```

这个命令可以让主节点退化为备份节点，并维持 60 秒。如果这段时间内没有新的主节点被选举出来，这个节点就可以要求重新进行选举。如果希望主节点退化为备份节点并持续更长（或者更短）的时间，可以自己指定时间（以秒为单位）：

```
> rs.stepDown(600) // 10分钟
```

12.3.2 阻止选举

如果需要对主节点做一些维护，但是不希望这段时间内将其他成员选举为主节点，那么可以在每个备份节点上执行 freeze 命令，以强制它们始终处于备份节点状态：

```
> rs.freeze(10000)
```

这个命令也会接受一个以秒表示的时间，表示在多长时间内保持备份节点状态。

维护完成之后，如果想"释放"其他成员，可以再次执行 freeze 命令，将时间指定为 0 即可：

```
> rs.freeze(0)
```

这样，其他成员就可以在必要时申请被选举为主节点。

也可以在主节点上执行 rs.freeze(0)，这样可以将退位的主节点重新变为主节点。

12.3.3 使用维护模式

当在副本集成员上执行某个非常耗时的操作时，这个成员就人进入维护模式（maintenance mode）：强制成员进入 RECOVERING 状态。有时，成员会自动进入维护模式，比如在成员上做压缩时。压缩开始之后，成员会进入 RECOVERING 状态，这样就不会有读请求发送给这个成员。客户端会停止从这个成员读取数据（如果之前有从这个成员读数据的话），这个成员也不能再作为复制源。

也可以通过执行 `replSetMaintenanceMode` 命令强制一个成员进入维护模式。如果一个成员远远落后于主节点，你不希望它继续处理读请求时，可以强制让这个成员进入维护模式。例如，下面这个脚本会自动检测成员是否落后于主节点 30 秒以上，如果是，就强制将这个成员转入维护模式：

```
function maybeMaintenanceMode() {
    var local = db.getSisterDB("local");

    // 如果成员不是备份节点（它可能是主节点
    // 或者已经处于维护状态），就直接返回
    if (!local.isMaster().secondary) {
        return;
    }

    // 查找这个成员最后一次操作的时间
    var last = local.oplog.rs.find().sort({"$natural" : -1}).next();
    var lastTime = last['ts']['t'];

    // 如果落后主节点 30 秒以上
    if (lastTime < (new Date()).getTime()-30) {
        db.adminCommand({"replSetMaintenanceMode" : true});
    }
    };
```

将成员从维护模式中恢复，可以使用如下命令：

```
> db.adminCommand({"replSetMaintenanceMode" : false});
```

12.4　监控复制

监控副本集的状态非常重要：不仅要监控是否所有成员都可用，也要监控每个成员处于什么状态，以及每个成员的数据新旧程度。有多个命令可以用来查看副本集相关信息。MMS（详见第 21 章）也维护着一些与复制相关的信息。

与复制相关的故障通常都是很短暂的：一个服务器刚才还连接不到另一个服务器，但是现在又可以连上了。要查看这样的问题，最简单的方式就是查看日志。确保自己知道日志的保存位置（而且*真的*被保存下来），确保能够访问到它们。

12.4.1　获取状态

`replSetGetStatus` 是一个非常有用的命令，可以返回副本集中每个成员的当前信息（这里的“当前”是从每个成员自身的角度来说的）。这个命令还有一个对应的辅助函数 `rs.status`：

```
> rs.status()
{
```

```
    "set" : "spock",
    "date" : ISODate("2012-10-17T18:17:52Z"),
    "myState" : 2,
    "syncingTo" : "server-1:27017",
    "members" : [
        {
            "_id" : 0,
            "name" : "server-1:27017",
            "health" : 1,
            "state" : 1,
            "stateStr" : "PRIMARY",
            "uptime" : 74824,
            "optime" : { "t" : 1350496621000, "i" : 1 },
            "optimeDate" : ISODate("2012-10-17T17:57:01Z"),
            "lastHeartbeat" : ISODate("2012-10-17T17:57:00Z"),
            "pingMs" : 3,
        },
        {
            "_id" : 1,
            "name" : "server-2:27017",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 161989,
            "optime" : { "t" : 1350377549000, "i" : 500 },
            "optimeDate" : ISODate("2012-10-17T17:57:00Z"),
            "self" : true
        },
        {
            "_id" : 2,
            "name" : "server-3:27017",
            "health" : 1,
            "state" : 3,
            "stateStr" : "RECOVERING",
            "uptime" : 24300,
            "optime" : { "t" : 1350411407000, "i" : 739 },
            "optimeDate" : ISODate("2012-10-16T18:16:47Z"),
            "lastHeartbeat" : ISODate("2012-10-17T17:57:01Z"),
            "pingMs" : 12,
            "errmsg" : "still syncing, not yet to minValid optime 507e9a30:851"
        }
    ],
    "ok" : 1
}
```

下面分别介绍几个最有用的字段。

- `self`

  这个字段只会出现在执行 `rs.status()` 函数的成员信息中，在本例中是 server-2。

- `stateStr`

  用于描述服务器状态的字符串。关于成员不同状态的描述，可以查看 10.2.1 节。

- `uptime`
  从成员可达一直到现在所经历的时间，单位是秒。对于 `"self"` 成员，这个值是从成员启动一直到现在的时间。因此，server-2 已经启动 161 989 秒了（大约 45 小时）。server-1 在过去的 21 小时中一直处于可用状态，server-3 在过去 7 小时中一直处于可用状态。

- `optimeDate`
  每个成员的 `oplog` 中最后一个操作发生的时间（也就是操作被同步过来的时间）。注意，这里的状态是每个成员通过心跳报告上来的状态，所以 `optime` 跟实际时间可能会有几秒钟的偏差。

- `lastHeartbeat`
  当前服务器最后一次收到其他成员心跳的时间。如果网络故障或者当前服务器比较繁忙，这个时间可能会是 2 秒钟之前。

- `pingMs`
  心跳从当前服务器到达某个成员所花费的平均时间，可以根据这个字段选择从哪个成员进行同步。

- `errmsg`
  成员在心跳请求中返回的状态信息。这个字段的内容通常只是一些状态信息，而不是错误信息。例如，server-3 的 `"errmsg"` 字段表示它正处于初始化同步过程中。这里的十六进制数字 507e9a30:851 是某个操作对应的时间戳，server-3 至少要同步完这个操作才能完成同步过程。

有几个字段的信息是重复的：`"state"` 与 `"stateStr"` 都表示成员的状态，只是 `"state"` 的值是状态的内部表示法。`"health"` 仅仅表示给定的服务器是否可达（可达是 1，不可达是 0），而从 `"state"` 和 `"stateStr"` 字段也可以得到这样的信息（如果服务器不可达，它们的值会是 `UNKNOWN` 或者 `DOWN`）。类似地，`"optime"` 和 `"optimeDate"` 的值也是相同的，只是表示方式不同：一个是用从新纪元开始的毫秒数表示的，另一个用一种更适合阅读的方式表示。

注意，这份报告是以执行 `rs.status()` 命令的成员的角度得出的：由于网络故障，这份报告可能不准确或者有些过时。

12.4.2 复制图谱

如果在备份节点上运行 `rs.status()`，输出信息中会有一个名为 `"syncingTo"` 的顶级字段，用于表示当前成员正在从哪个成员处进行复制。如果在每个成员上运

行 `replSetGetStatus` 命令，就可以弄清楚复制图谱（replication graph）。假设 `server1` 表示连接到 server1 的数据库连接，`server2` 表示连接到 server2 的数据库连接，以此类推，然后分别在这些连接上执行下面的命令：

```
> server1.adminCommand({replSetGetStatus: 1})['syncingTo']
server0:27017
> server2.adminCommand({replSetGetStatus: 1})['syncingTo']
server1:27017
> server3.adminCommand({replSetGetStatus: 1})['syncingTo']
server1:27017
> server4.adminCommand({replSetGetStatus: 1})['syncingTo']
server2:27017
```

所以，server0 是 server1 的同步源，server1 是 server2 和 server3 的同步源，server2 是 server4 的同步源。

MongoDB 根据 ping 时间选择同步源。一个成员向另一个成员发送心跳请求，就可以知道心跳请求所耗费的时间。MongoDB 维护着不同成员间请求的平均花费时间。选择同步源时，会选择一个离自己比较近而且数据比自己新的成员（所以，不会出现循环复制的问题，每个成员要么从主节点复制，要么从数据比它新的成员处复制）。

因此，如果在备份数据中心中添加一个新成员，它很可能会从与自己同在一个数据中心内的其他成员处复制，而不是从位于另一个数据中心的主节点处复制（这样可以减少网络流量），如图 12-1 所示。

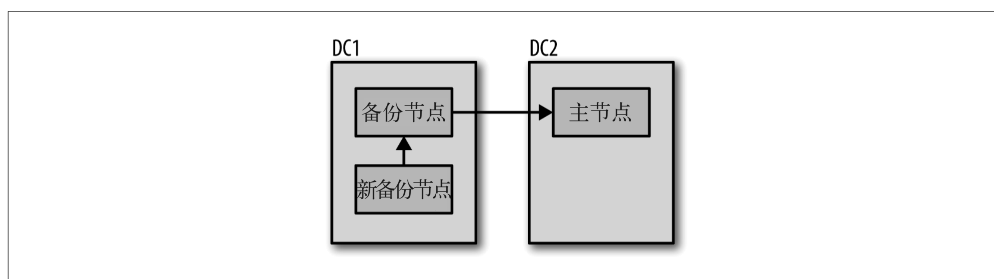


图 12-1：新的备份节点通常会从与自己处于同一个数据中心的其他成员进行复制

但是，自动复制链（automatic replication chaining）也有一些缺点：复制链越长，将写操作复制到所有服务器所花费的时间就越长。假设所有服务器都位于同一个数据中心内，然后，由于网络速度异常，新添加一个成员之后，MongoDB 的复制链如图 12-2 所示。

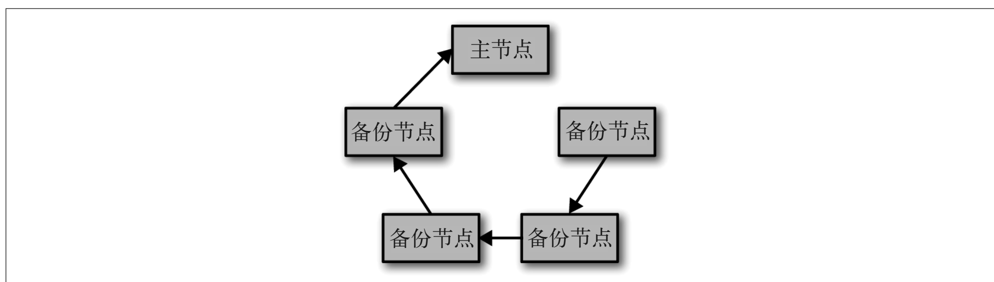


图 12-2：复制链越长，将数据同步到全部服务器花费的时间就越长

通常不太可能发生这样的情况，但是并非不可能。但这种情况通常是不可取的：复制链中的每个备份节点都要比它前面的备份节点稍微落后一点。只要出现这种状况，可以用 replSetSyncFrom（或者是它对应的辅助函数 rs.syncFrom()）命令修改成员的复制源进行修复。

连接到需要修改复制源的备份节点，运行这个命令，为其指定一个复制源：

```
> secondary.adminCommand({"replSetSyncFrom" : "server0:27017"})
```

可能要花费几秒钟的时间才能切换到新的复制源。如果在这个成员上再次执行 rs.status()，会发现 "syncingTo" 字段的值已经变成了 "server0:27017"。

现在，server4 会一直从 server0 进行复制，直到 server0 不可用或者远远落后于其他成员为止。

12.4.3 复制循环

如果复制链中出现了环，那么就称为发生了复制循环。例如，A 从 B 处同步数据，B 从 C 处同步数据，C 从 A 处同步数据，这就是一个复制循环。因为复制循环中的成员都不可能成为主节点，所以这些成员无法复制新的写操作，就会越来越落后。另一方面，如果每个成员都是自动选取复制源，那么复制循环是不可能发生的。

但是，使用 replSetSyncFrom 强制为成员设置复制源时，就可能会出现复制循环。在手动修改成员的复制源时，应该仔细查看 rs.status() 的输出信息，避免造成复制循环。当用 replSetSyncFrom 为成员指定一个并不比它领先的成员作为复制源时，系统会给出警告，但是仍然允许这么做。

12.4.4 禁用复制链

当一个备份节点从另一个备份节点（而不是主节点）复制数据时，就会形成复制链。前面说过，成员会自动选择其他成员作为复制源。

可以禁用复制链，强制要求每个成员都从主节点进行复制，只需要将 "allowChaining" 设置为 false 即可（如果不指定这个选项，默认是 true）：

```
> var config = rs.config()
> // 如果设置子对象不存在，就自动创建一个空的
> config.settings = config.settings || {}
> config.settings.allowChaining = false
> rs.reconfig(config)
```

将 allowChaining 设置为 false 之后，所有成员都会从主节点复制数据。如果主节点变得不可用，那么各个成员就会从其他备份节点处复制数据。

12.4.5　计算延迟

跟踪复制情况的一个重要指标是备份节点与主节点之间的延迟程度。延迟（lag）是指备份节点相对于主节点的落后程度，是主节点最后一次操作的时间戳与备份节点最后一次操作的时间戳的差。

可以使用 rs.status() 查看成员的复制状态，也可以通过在主节点上执行 db.printReplicationInfo()（这个命令的输出信息中包括 oplog 相关信息），或者在备份节点上执行 db.printSlaveReplicationInfo() 快速得到一份摘要信息。注意，这两个都是 db 的函数，而不是 rs 的。

db.printReplicationInfo 的输出中包括主节点的 oplog 信息：

```
> db.printReplicationInfo();
    conÑgured oplog size:    10.48576MB
    log length start to end: 34secs (0.01hrs)
    oplog Ñrst event time:   Tue Mar 30 2010 16:42:57 GMT-0400 (EDT)
    oplog last event time:   Tue Mar 30 2010 16:43:31 GMT-0400 (EDT)
    now:                     Tue Mar 30 2010 16:43:37 GMT-0400 (EDT)
```

上面的输出信息中包含了 oplog 的大小，以及 oplog 中包含的操作的时间范围。在本例中，oplog 的大小大约是 10 MB，而且只包含大约最近 30 秒的操作。

在实际的部署中，oplog 会大得多（12.4.6 节会讲述如何修改 oplog 的大小）。我们希望 oplog 的长度至少要能够容纳一次完整同步的所有操作。这样，备份节点就不会在完成初始化同步之前与 oplog 脱节。



oplog 中第一条操作与最后一条操作的时间差就是操作日志的长度。如果服务器才刚刚启动，刚启动时的 oplog 是空的，那么 oplog 中的第一条操作会距离现在非常近。在这种情况下，日志长度会比较小，即使 oplog 仍然有可用空间。对于那些已经运行了比较长的时间，oplog 已经至少被填满一次的服务器来说，日志长度是一个非常有用的度量指标。

在备份节点上运行 `db.printSlaveReplicationInfo()`，可以得到当前成员的复制源，以及当前成员相对复制源的落后程度等信息：

```
> db.printSlaveReplicationInfo();
    source:   server-0:27017
    syncedTo: Tue Mar 30 2012 16:44:01 GMT-0400 (EDT)
    = 12secs ago (0hrs)
```

这样就可以知道当前成员正在从哪个成员处复制数据。在这个例子中，备份节点比主节点落后 12 秒。

注意，副本集成员的延迟是相对于主节点来说的，而不是表示需要多长时间才能更新到最新。在一个写操作非常少的系统中，有可能会造成延迟过大的幻觉。假设一小时执行一次写操作。刚刚执行完这次写操作之后，复制之前，备份节点会落后于主节点一小时。但是，只需要几毫秒时，备份节点就可以追上主节点。当监控低吞吐量的系统时，这个值可能会造成迷惑。

12.4.6 调整oplog大小

可以将主节点的 oplog 长度看作维护工作的时间窗。如果主节点的 oplog 长度是一小时，那么你就只有一小时的时间可以用于修复各种错误，不然的话备份节点可能会落后于主节点太多，导致不得不重新进行完全同步。所以，你通常可能希望 oplog 能够保存几天或者一个星期的数据，从而给自己预留足够的时间，用于处理各种突发状况。

可惜，在 oplog 被填满之前很难知道它的长度，也没有办法在服务器运行期间调整 oplog 大小。但是，可以依次将每台服务器下线，调整它的 oplog，然后重新把它添加到副本集中。记住，每一个可能成为主节点的服务器都应该拥有足够大的 oplog，以预留足够的时间窗用于进行维护。

如果要增加 oplog 大小，可以按照如下步骤。

(1) 如果当前服务器是主节点，让它退位，以便让其他成员的数据能够尽快更新到与它一致。

(2) 关闭当前服务器。

(3) 将当前服务器以单机模式启动。

(4) 临时将 oplog 中的最后一条 `insert` 操作保存到其他集合中：

```
> use local
> // op:"i" 用于查找最后一条 insert 操作
> var cursor = db.oplog.rs.find({"op" : "i"})
```

```
> var lastInsert = cursor.sort({"$natural" : -1}).limit(1).next()
> db.tempLastOp.save(lastInsert)
>
> // 确保保存成功，这非常重要!
> db.tempLastOp.findOne()
```

也可以使用最后一项 `update` 或者 `delete` 操作，但是 `$` 操作符不能插入到集合中。

(5) 删除当前的 oplog：

```
> db.oplog.rs.drop()
```

(6) 创建一个新的 oplog：

```
> db.createCollection("oplog.rs", {"capped" : true, "size" : 10000})
```

(7) 将最后一条操作记录写回 oplog：

```
> var temp = db.tempLastOp.findOne()
> db.oplog.rs.insert(temp)
>
> // 要确保插入成功
> db.oplog.rs.findOne()
```

确保最后一条操作记录成功插入 oplog。如果没有插入成功，把当前服务器添加到副本集之后，它会删除所有数据，然后重新进行一次完整同步。

(8) 最后，将当前服务器作为副本集成员重新启动。注意，由于这时它的 oplog 只有一条记录，所以在一段时间内无法知道 oplog 的真实长度。另外，这个服务器现在也并不适合作为其他成员的复制源。

通常不应该减小 oplog 的大小：即使 oplog 可能会有几个月那么长，但是通常总是有足够的硬盘空间来保存 oplog，oplog 并不会占用任何珍贵的资源（比如 CPU 或 RAM）。

12.4.7 从延迟备份节点中恢复

假设有人不小心删除了一个数据库，幸好你有一个延迟备份节点。现在，需要放弃其他成员的数据，明确将延迟备份节点指定为数据源。有几种方法可以使用。

下面介绍最简单的方法。

(1) 关闭所有其他成员。

(2) 删除其他成员数据目录中的所有数据。确保每个成员（除了延迟备份节点）的数据目录都是空的。

(3) 重启所有成员，然后它们会自动从延迟备份节点中复制数据。

这种方式非常简单，但是，在其他成员完成初始化同步之前，副本集中将只有一个成员可用（延迟备份节点）而且这个成员很可能会过载。

根据数据量的不同，第二种方式可能更好，也可能更差。

(1) 关闭所有成员，包括延迟备份节点。

(2) 删除其他成员（除了延迟备份节点）的数据目录。

(3) 将延迟备份节点的数据文件复制到其他服务器。

(4) 重启所有成员。

注意，这样会导致所有服务器都与延迟备份节点拥有同样大小的 oplog，这可能不是你想要的。

12.4.8　创建索引

如果向主节点发送创建索引的命令，主节点会正常创建索引，然后备份节点在复制“创建索引”操作时也会创建索引。这是最简单的创建索引的方式，但是创建索引是一个需要消耗大量资源的操作，可能会导致成员不可用。如果所有备份节点都在同一时间开始创建索引，那么几乎所有成员都会不可用，一直到索引创建完成。

因此，可能你会希望每次只在一个成员上创建索引，以降低对应用程序的影响。如果要这么做，有下面几个步骤。

(1) 关闭一个备份节点服务器。

(2) 将这个服务器以单机模式启动。

(3) 在单机模式下创建索引。

(4) 索引创建完成之后，将服务器作为副本集成员重新启动。

(5) 对副本集中的每个备份节点重复第 (1) 步 ~ 第 (4) 步。

现在副本集的每个成员（除了主节点）都已经成功创建了索引。现在你有两个选择，应该根据自己的实际情况选择一个对生产系统影响最小的方式。

(1) 在主节点上创建索引。如果系统会有一段负载比较小的“空闲期”，那会是非常好的创建索引的时机。也可以修改读取首选项，在主节点创建索引期间，将读操作发送到备份节点上。

  主节点创建索引之后，备份节点仍然会复制这个操作，但是由于备份节点中已

经有了同样的索引，实际上不会再次创建索引。

(2) 让主节点退化为备份节点，对这个服务器执行上面的 4 步。这时就会发生故障转移，在主节点退化为备份节点创建索引期间，会有新的节点被选举为主节点，保证系统正常运转。索引创建完成之后，可以重新将服务器添加到副本集。

注意，可以使用这种技术为某个备份节点创建与其他成员不同的索引。这种方式在做离线数据处理时会非常有用，但是，如果某个备份节点的索引与其他成员不同，那么它永远不能成为主节点：应该将它的优先级设为 0。

如果要创建唯一索引，需要先确保主节点中没有被插入重复的数据，或者应该首先为主节点创建唯一索引。否则，可能会有重复数据插入主节点，这会导致备份节点复制时出错，如果遇到这样的错误，备份节点会将自己关闭。你不得不以单机模式启动这台服务器，删除唯一索引，然后重新将其加入副本集。

12.4.9　在预算有限的情况下进行复制

如果预算有限，不能使用多台高性能服务器，可以考虑将备份节点只用于灾难恢复，这样的备份节点不需要太大的 RAM 和太好的 CPU，也不需要太高的磁盘 IO。这样，始终将高性能服务器作为主节点，比较便宜的服务器只用于备份，不处理任何客户端请求（将客户端配置为将全部读请求发送到主节点）。对于这样的备份节点，应该设置这些选项。

- `"priority" : 0`
  优先级为 0 的备份节点永远不会成为主节点。

- `"hidden" : true`
  将备份节点设为隐藏，客户端就无法将读请求发送给它了。

- `"buildIndexes" : false`
  这个选项是可选的，如果在备份节点上创建索引的话，会极大地降低备份节点的性能。如果不在备份节点上创建索引，那么从备份节点中恢复数据之后，需要重新创建索引。

- `"votes" : 0`
  在只有两台服务器的情况下，如果将备份节点的投票数设为 0，那么当备份节点挂掉之后，主节点仍然会一直是主节点，不会因为达不到“大多数”的要求而退位。如果还有第三台服务器（即使它是你的应用服务器），那么应该在第三台服务器上运行一个仲裁者成员，而不是将第三台服务器的投票数量设为 0。

在没有足够的预算购买多台高性能服务器的情况下，可以用这样的备份节点来保证系统和数据安全。

12.4.10 主节点如何跟踪延迟

作为其他成员的同步源的成员会维护一个名为 local.slaves 的集合，这个集合中保存着所有正从当前成员进行数据同步的成员，以及每个成员的数据新旧程度。如果使用 w 参数执行查询，MongoDB 会根据这些信息确定是否有足够多、足够新的备份节点可以用来处理查询。

local.slaves 集合实际上是内存中数据结构的“回声”，所以其中的数据可能会有几秒钟的延迟：

```
> db.slaves.find()
{ "_id" : ObjectId("4c1287178e00e93d1858567c"), "host" : "10.4.1.100",
  "ns" : "local.oplog.rs", "syncedTo" : { "t" : 1276282710000, "i" : 1 } }
{ "_id" : ObjectId("4c128730e6e5c3096f40e0de"), "host" : "10.4.1.101",
  "ns" : "local.oplog.rs", "syncedTo" : { "t" : 1276282710000, "i" : 1 } }
```

每个服务器的 "_id" 字段非常重要：它是所有正在从当前成员进行数据同步的服务器的标识符。连接到一个成员，然后查询 local.me 集合就可以知道一个成员的标识符：

```
> db.me.findOne()
{ "_id" : ObjectId("50e6edb517c789e46695212f"), "host" : "server-1" }
```

非常偶然的情况下，由于网络故障，可能会发现有多台服务器拥有相同的标识符。在这种情况下，只能知道其中一台服务器相对于主节点的新旧程度。所以，这可能会导致应用程序故障（如果应用程序需要等待特定数量的服务器完成写操作）和分片问题（数据迁移被复制到“大多数”备份节点之前，无法继续做数据迁移）。如果多台服务器拥有相同的 "_id"，可以依次登录到每台服务器，删除 local.me 集合，然后重新启动 mongod。启动时，mongod 会使用新的 "_id" 重新生成 local.me 集合。

如果服务器的地址发生了改变（假定 "_id" 没有变，但是主机名变了），可能会在本地数据库的日志中看到键重复异常（duplicate key exception）。遇到这种情况时，删除 local.slaves 集合即可（这比之前的例子简单，因为只需要清除旧数据即可，不需要处理数据冲突）。

mongod 不会清理 local.slaves 集合，所以，它可能会列出某个几个月之前就不再把该成员作为同步源的服务器（或者是已经不在副本集内的成员）。由于 MongoDB 只

是把这个集合用于报告复本集状态，所以这个集合中的过时数据并不会有什么影响。如果你觉得这个集合中的旧数据会造成困惑或者是过于混乱，可以将整个集合删除。几秒钟之后，如果有新的服务器将当前成员作为复制源的话，这个集合就会重新生成。

如果备份节点之间形成了复制链，你可能会注意到某个特定的服务器在主节点的local.slaves 集合中有多个文档。这是因为，每个备份节点都会将复制请求转发给它的复制源，这样主节点就能够知道每个备份节点的同步源。这称为"影同步"(ghost syncs)，因为这些请求并不会要求进行数据同步，只是把每个备份节点的同步源报告给主节点。



local 数据库只用于维护复制相关信息，它并不会被复制。因此，如果希望某些数据只存在于特定的机器上，可以将这些数据保存在 local 数据库的集合中。

12.5 主从模式

MongoDB 最初支持一种比较传统的主从模式（master-slave），在这种模式下，MongoDB 不会做自动故障转移，而且需要明确声明主节点和从节点。有两种情形应该使用主从模式而不是副本集：需要多于 11 个备份节点，或者是需要复制单个数据库。除非迫不得已，否则都应该使用副本集。副本集更易维护，而且功能齐全。主从模式以后会被废弃，当副本集能够支持无限数据的成员时，主从模式很可能会被立即废弃。

但是，有时可能确实需要 11 台以上的备份节点（从节点），或者是需要复制单个数据库。这些情况下，应该使用主从模式。

如果要将服务器设为主节点，可以使用 `--master` 选项启动服务器。对于从节点，有两个可用的选项：`--slave` 和 `--source` *`master`*。`--source` 用于指定同步源的主机名和端口号。注意，不要使用 `--replSet` 选项，因为现在是要设置主从模式，而不是副本集。

假如有两台服务器，`server-0` 和 `server-1`，可以这么做：

```
$ # server-0
$ mongod --master
$
$ # server-1
$ mongod --slave --source server-0:27017
```

这样，主从模式就设置成功了，不需要其他的设置。在主节点执行的写操作，会被复制到从节点上。

主从模式也可以用于复制单个数据库。可以使用 `--only` 选项选择需要进行复制的数据库。

```
$ mongod --slave --source server-1:27017 --only super-important-db
```

驱动程序不会自动将读请求发送给从节点。如果要从从节点读取数据，需要显式地创建一个连接到从节点的数据库连接。

12.5.1　从主从模式切换到副本集模式

从主从模式切换到副本集模式，需要停机一段时间，步骤如下。

(1) 停止系统的所有写操作。这非常重要，因为在主从模式下，从节点并不会维护一份 oplog，所以它无法将升级期间落下的操作同步过来。

(2) 关闭所有的 mongod 服务器。

(3) 使用 `--replSet` 选项重启主节点，不再使用 `--master`。

(4) 初始化这个只有一个成员的副本集，这个成员会成为副本集中的主节点。

(5) 使用 `--replSet` 和 `--fastsync` 选项启动从节点。通常，如果向副本集中添加一个没有 oplog 的成员，这个成员会立即进入完全的初始化同步过程。`fastsync` 选项用于告诉新成员不会担心 oplog 的问题，直接从主节点最新的操作开始同步即可。

(6) 使用 `rs.add()` 将之前的从节点加入副本集。

(7) 对每个从节点，重复第 (5) 步和第 (6) 步。

(8) 当所有从节点都变为备份节点之后，就可以开启系统的写功能了。

(9) 从配置文件、命令行别名和内存中删除 `fastsync` 选项。这是一个非常危险的选项，它会使成员启动时跳过一些需要同步的操作。只有在从主从模式切换到副本集时才可以使用这个命令。现在已经切换完成了，不再需要这个选项了。

现在，主从模式已经被切换为副本集了。

12.5.2　让副本集模仿主从模式的行为

通常你会希望主节点长时间可用，因此，万一主节点不可用，应该允许自动故障转移。但是，对于某些副本集，你可能会要求手动选择新的主节点，不允许进行自动故障转移。这样的话，副本集的行为就跟主从模式一样了（对于这种情况，建议使

用主从模式，而不是使用副本集）。

为了实现这个目的，需要重新配置副本集，将所有成员（除主节点之外）的 `priority` 和 `votes` 设为 0。这样一来，如果主节点挂了，不会有任何成员寻求被选举为主节点。另外，如果所有备份节点都挂了，主节点也仍然会一直保持主节点状态，不会退位（因为它是整个系统中唯一一个拥有投票权的成员）。

下面的配置文件会创建一个具有 5 个成员的副本集，其中 server-0 会始终作为主节点，其他 4 个成员会始终作为备份节点：

```
{
    "_id" : "spock",
    "members" : [
        {"_id" : 0, "host" : "server-0:27017"},
        {"_id" : 1, "host" : "server-1:27017", "priority" : 0, "votes" : 0},
        {"_id" : 2, "host" : "server-2:27017", "priority" : 0, "votes" : 0},
        {"_id" : 3, "host" : "server-3:27017", "priority" : 0, "votes" : 0},
        {"_id" : 4, "host" : "server-4:27017", "priority" : 0, "votes" : 0}
    ]
}
```

如果主节点挂了，管理员必须手动选出新的主节点。

如果要手动将某个备份节点提升为主节点，首先要连接到这个备份节点，然后执行强制重新配置，将它的 `priority` 和 `votes` 修改为 1，同时将先前的主节点的 `priority` 和 `votes` 修改为 0。

例如，如果 server-0 挂了，可以连接到希望提升为新的主节点的备份节点（比如 server-1），然后以下面的方式修改配置：

```
> var config = rs.config()
> config.members[1].priority = 1
> config.members[1].votes = 1
> config.members[0].priority = 0
> config.members[0].votes = 0
> rs.reconfig(config, {"force" : true})
```

现在，如果运行 `rs.config()`，就可以看到新的副本集配置信息了：

```
> rs.conÑg()
{
    "_id" : "spock",
    "version" : 3
    "members" : [
        {"_id" : 0, "host" : "server-0:27017", "priority" : 0, "votes" : 0},
        {"_id" : 1, "host" : "server-1:27017"},
        {"_id" : 2, "host" : "server-2:27017", "priority" : 0, "votes" : 0},
        {"_id" : 3, "host" : "server-3:27017", "priority" : 0, "votes" : 0},
```

```
            {"_id" : 4, "host" : "server-4:27017", "priority" : 0, "votes" : 0}
        ]
    }
```

如果新的主节点又挂了，可以重复上面的步骤，手工将某个备份节点提升为新的主节点。

第四部分

分片

第 13 章 分片

本章介绍如果扩展 MongoDB：

- 分片和集群组件；
- 如何配置分片；
- 分片与应用程序的交互 。

13.1 分片简介

分片（sharding）是指将数据拆分，将其分散存放在不同的机器上的过程。有时也用分区（partitioning）来表示这个概念。将数据分散到不同的机器上，不需要功能强大的大型计算机就可以储存更多的数据，处理更大的负载。

几乎所有数据库软件都能进行手动分片（manual sharding）。应用需要维护与若干不同数据库服务器的连接，每个连接还是完全独立的。应用程序管理不同服务器上不同数据的存储，还管理在合适的数据库上查询数据的工作。这种方法可以很好地工作，但是非常难以维护，比如向集群添加节点或从集群删除节点都很困难，调整数据分布和负载模式也不轻松。

MongoDB 支持自动分片（autosharding），可以使数据库架构对应用程序不可见，也可以简化系统管理。对应用程序而言，好像始终在使用一个单机的 MongoDB 服务器一样。另一方面，MongoDB 自动处理数据在分片上的分布，也更容易添加和删除分片。

不管从开发角度还是运营角度来说，分片都是最困难最复杂的 MongoDB 配置方式。

有很多组件可以用于自动配置、监控和数据转移。在尝试部署或使用分片集群之前，你需要先熟悉前面章节中讲过的单机服务器和副本集。

13.2　理解集群的组件

MongoDB 的分片机制允许你创建一个包含许多台机器（分片）的集群，将数据子集分散在集群中，每个分片维护着一个数据集合的子集。与单机服务器和副本集相比，使用集群架构可以使应用程序具有更大的数据处理能力。



许多人可能会混淆复制和分片的概念。记住，复制是让多台服务器都拥有同样的数据副本，每一台服务器都是其他服务器的镜像，而每一个分片都有其他分片拥有不同的数据子集。

分片的目标之一是创建一个拥有 5 台、10 台甚至 1000 台机器的集群，整个集群对应用程序来说就像是一台单机服务器。为了对应用程序隐藏数据库架构的细节，在分片之前要先执行 `mongos` 进行一次路由过程。这个路由服务器维护着一个“内容列表”，指明了每个分片包含什么数据内容。应用程序只需要连接到路由服务器，就可以像使用单机服务器一样进行正常的请求了，如图 13-1 所示。路由服务器知道哪些数据位于哪个分片，可以将请求转发给相应的分片。每个分片对请求的响应都会发送给路由服务器，路由服务器将所有响应合并在一起，返回给应用程序。对应用程序来说，它只知道自己是连接到了一台单机 mongod 服务器，如图 13-2 所示。

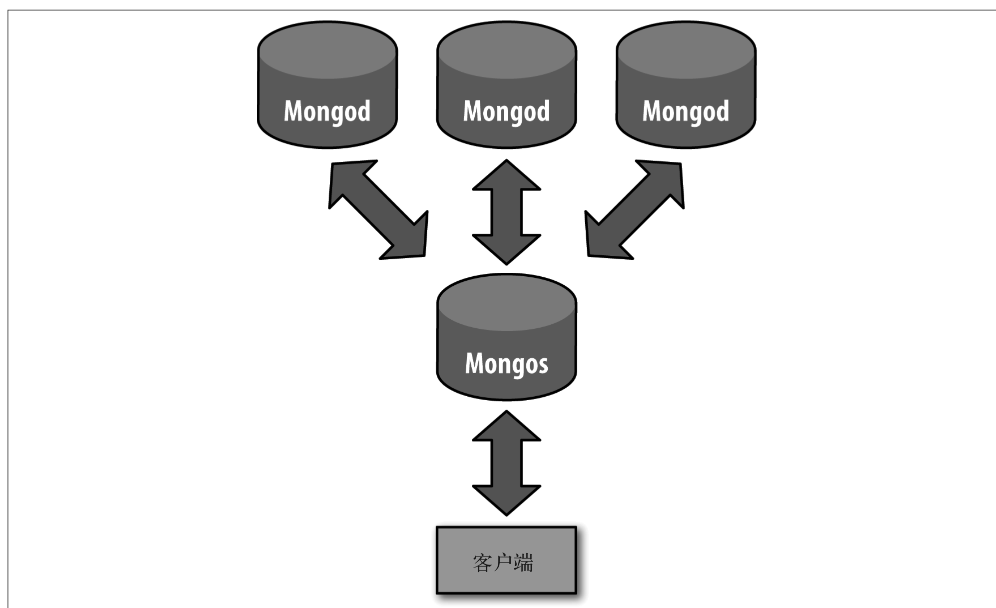


图 13-1：使用分片的连接

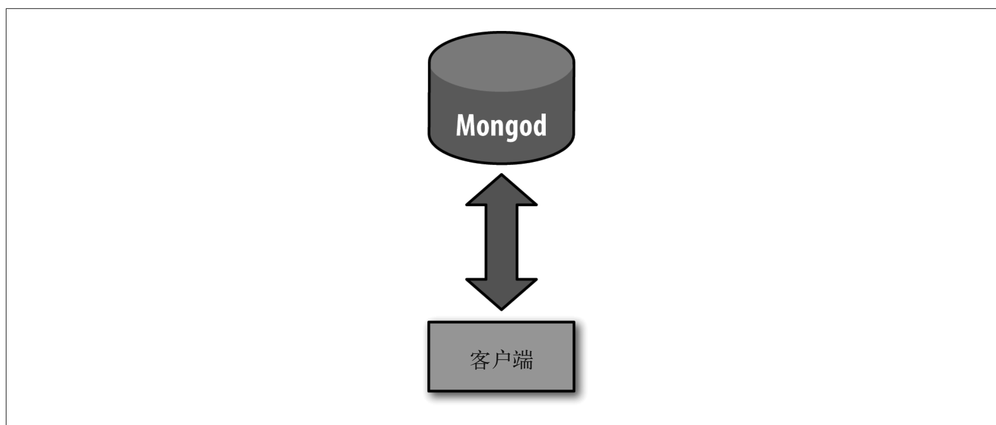


图 13-2：不使用分片的连接

13.3 快速建立一个简单的集群

如前面介绍复制时一样，本节会在单台服务器上快速建立一个集群。首先，使用 `--nodb` 选项启动 mongo shell：

```
$ mongo --nodb
```

使用 `ShardingTest` 类创建集群：

```
> cluster = new ShardingTest({"shards" : 3, "chunksize" : 1})
```

第 16 章会详细介绍 `chunksize` 选项，目前来说可以简单将其设置为 1。

运行这个命令就会创建一个包含 3 个分片（mongod 进程）的集群，分别运行在 30000、30001、30002 端口。默认情况下，`ShardingTest` 会在 30999 端口启动 mongos。接下来就连接到这个 mongos 开始使用集群。

集群会将日志输出到当前 shell 中，所以再打开一个 shell 用来连接到集群的 mongos：

```
> db = (new Mongo("localhost:30999")).getDB("test")
```

现在的情况如图 13-1 所示：客户端（shell）连接到了一个 mongos。现在就可以将请求发送给 mongos 了，它会自动将请求路由到合适的分片。客户端不需要知道分片的任何信息，比如分片数量和分片地址。只要有分片存在，就可以向 mongos 发送请求，它会自动将请求转发到合适的分片上。

首先插入一些数据：

```
> for (var i=0; i<100000; i++) {
...     db.users.insert({"username" : "user"+i, "created_at" : new Date()});
... }
> db.users.count()
100000
```

可以看到，与 mongos 进行交互与使用单机服务器完全一样，如图 13-2 所示。

运行 `sh.status()` 可以看到集群的状态：分片摘要信息、数据库摘要信息、集合摘要信息：

```
> sh.status()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
        { "_id" : "shard0000", "host" : "localhost:30000" }
        { "_id" : "shard0001", "host" : "localhost:30001" }
        { "_id" : "shard0002", "host" : "localhost:30002" }
  databases:
        { "_id" : "admin", "partitioned" : false, "primary" : "conǸg" }
        { "_id" : "test", "partitioned" : false, "primary" : "shard0001" }
```

`sh` 命令与 `rs` 命令很像，除了它是用于分片的：`rs` 是一个全局变量，其中定义了许多分片操作的辅助函数。可以运行 `sh.help()` 查看可以使用的辅助函数。如 `sh.stats()` 的输出所示，当前拥有 3 个分片，2 个数据库（其中 admin 数据库是自动创建的）。

与上面 `sh.stats()` 的输出信息不同，test 数据库可能有一个不同的主分片（primary shard）。主分片是为每个数据库随机选择的，所有数据都会位于主分片上。MongoDB 现在还不能自动将数据分发到不同的分片上，因为它不知道你希望如何分发数据。必须要明确指定，对于每一个集合，应该如何分发数据。



主分片与副本集中的主节点不同。主分片指的是组成分片的整个副本集。而副本集中的主节点是指副本集中能够处理写请求的单台服务器。

要对一个集合分片，首先要对这个集合的数据库启用分片，执行如下命令：

```
> sh.enableSharding("test")
```

现在就可以对 test 数据库内的集合进行分片了。

对集合分片时，要选择一个片键（shard key）。片键是集合的一个键，MongoDB 根据这个键拆分数据。例如，如果选择基于 "username" 进行分片，MongoDB 会根据不同的用户名进行分片："a1-steak-sauce" 到 "defcon" 位于第一片，"defcon1" 到 "howie1998" 位于第二片，以此类推。选择片键可以认为是选择集合中数据的顺序。它与索引是个相似的概念：随着集合的不断增长，片键就会成为集合上最重要的索引。只有被索引过的键才能够作为片键。

在启用分片之前，先在希望作为片键的键上创建索引：

```
> db.users.ensureIndex({"username" : 1})
```

现在就可以依据 "username" 对集合分片了：

```
> sh.shardCollection("test.users", {"username" : 1})
```

尽管我们这里选择片键时并没有作太多考虑，但是在实际中应该仔细斟酌。第 15 章会详细介绍如何选择片键。

几分钟之后再次运行 sh.status()，可以看到，这次的输出信息比较多：

```
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
    {  "_id" : "shard0000", "host" : "localhost:30000" }
    {  "_id" : "shard0001", "host" : "localhost:30001" }
    {  "_id" : "shard0002", "host" : "localhost:30002" }
  databases:
    { "_id" : "admin", "partitioned" : false, "primary" : "config" }
    { "_id" : "test", "partitioned" : true, "primary" : "shard0000" }
       test.users chunks:
           shard0001    4
           shard0002    4
           shard0000    5
       { "username" : { $minKey : 1 } } -->> { "username" : "user1704" }
           on : shard0001
       { "username" : "user1704" } -->> { "username" : "user24083" }
           on : shard0002
       { "username" : "user24083" } -->> { "username" : "user31126" }
           on : shard0001
       { "username" : "user31126" } -->> { "username" : "user38170" }
           on : shard0002
       { "username" : "user38170" } -->> { "username" : "user45213" }
           on : shard0001
       { "username" : "user45213" } -->> { "username" : "user52257" }
           on : shard0002
```

```
{ "username" : "user52257" } -->> { "username" : "user59300" }
    on : shard0001
{ "username" : "user59300" } -->> { "username" : "user66344" }
    on : shard0002
{ "username" : "user66344" } -->> { "username" : "user73388" }
    on : shard0000
{ "username" : "user73388" } -->> { "username" : "user80430" }
    on : shard0000
{ "username" : "user80430" } -->> { "username" : "user87475" }
    on : shard0000
{ "username" : "user87475" } -->> { "username" : "user94518" }
    on : shard0000
{ "username" : "user94518" } -->> { "username" : { $maxKey : 1 } }
    on : shard0000
```

集合被分为了多个数据块，每一个数据块都是集合的一个数据子集。这些是按照片键的范围排列的（`{"username" : `*`minValue`*`} -->> {"username" : `*`maxValue`*`}` 指出了每个数据块的数据范围）。通过查看输出信息中的 `"on" : shard` 部分，可以发现集合数据比较均匀地分布在不同分片上。

将集合拆分为多个数据块的过程如图 13-3 到图 13-5 所示。在分片之前，集合实际上是一个单一的数据块。分片依据片键将集合拆分为多个数据块，如图 13-4 所示。这块数据块被分布在集群中的每个分片上，如图 13-5 所示。

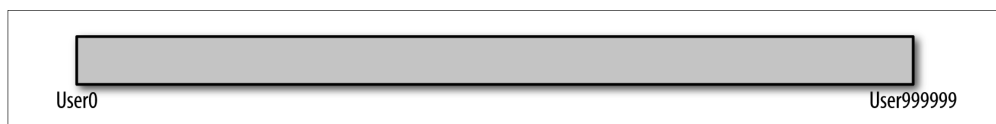


图 13-3：在分片之前，可以认为集合是一个单一的数据块，从片键的最小值一直到片键的最大值都位于这个块

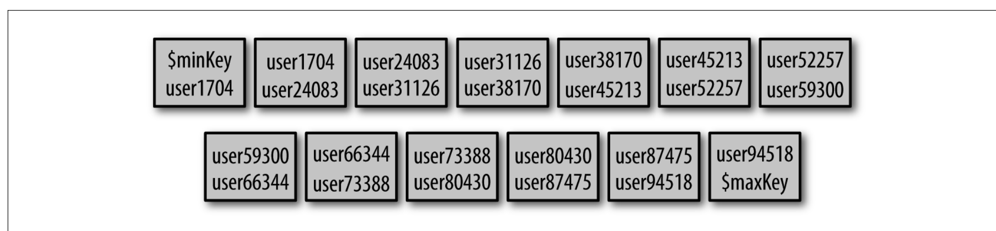


图 13-4：分片依据片键范围将集合拆分为多个数据块

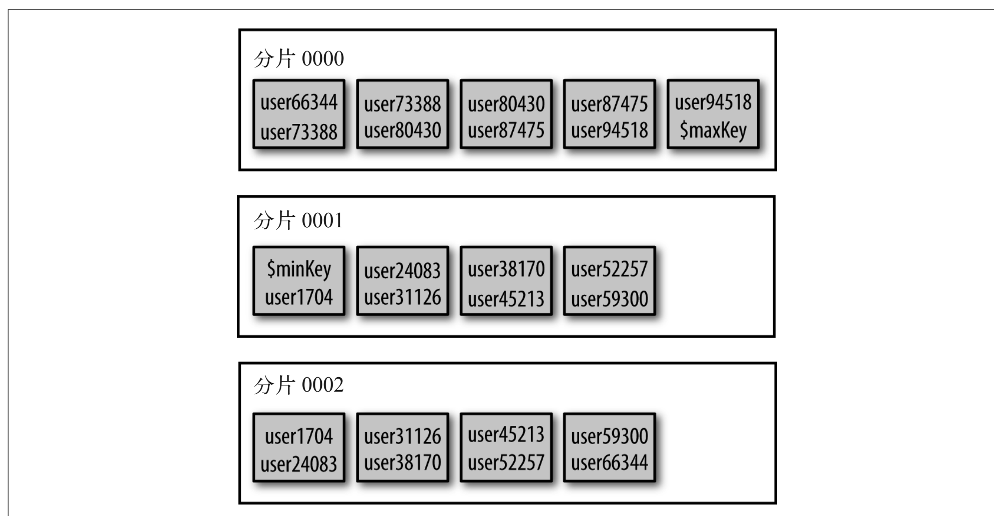


图 13-5：数据块均衡地分布在不同分片上

注意，数据块列表开始的键值和结束的键值：`$minkKey` 和 `$maxKey`。可以将 `$minKey` 认为是“负无穷”，它比 MongoDB 中的任何值都要小。类似地，可以将 `$maxKey` 认为是“正无穷”，它比 MongoDB 中的任何值都要大。因此，经常会见到这两个“端值”出现在数据块范围中。片键值的范围始终位于 `$minKey` 和 `$maxKey` 之间。这些值实际上是 BSON 类型，只是用于内部使用，不应该被用在应用程序中。如果希望在 shell 中使用的话，可以用 `MinKey` 和 `MaxKey` 常量代替。

现在数据已经分布在多个分片上了，接下来做一些查询操作。首先，做一个基于指定的用户名的查询：

```
> db.users.ﬁnd({username: "user12345"})
{
    "_id" : ObjectId("50b0451951d30ac5782499e6"),
    "username" : "user12345",
    "created_at" : ISODate("2012-11-24T03:55:05.636Z")
}
```

可以看到，查询可以正常工作。现在运行 `explain()` 来看看 MongoDB 到底是如何处理这次查询的：

```
> db.users.find({username: "user12345"}}).explain()
{
  "clusteredType" : "ParallelSort",
  "shards" : {
    "localhost:30001" : [
      {
        "cursor" : "BtreeCursor username_1",
```

```
          "nscanned" : 1,
          "nscannedObjects" : 1,
          "n" : 1,
          "millis" : 0,
          "nYields" : 0,
          "nChunkSkips" : 0,
          "isMultiKey" : false,
          "indexOnly" : false,
          "indexBounds" : {
            "username" : [
              [
                "user12345",
                "user12345"
              ]
            ]
          }
        }
      ]
    },
    "n" : 1,
    "nChunkSkips" : 0,
    "nYields" : 0,
    "nscanned" : 1,
    "nscannedObjects" : 1,
    "millisTotal" : 0,
    "millisAvg" : 0,
    "numQueries" : 1,
    "numShards" : 1
  }
```

输出信息包含两个部分：一个看起来比较普通的 `explain()` 输出嵌套在另一个 `explain()` 输出中。外层的 `explain()` 输出来自 mongos：描述了为了处理这个查询，mongos 所做的工作。内层的 `explain()` 输出来自查询所使用的分片，在本例中是 `localhost:30001`。

由于 `"username"` 是片键，所以 mongos 能够直接将查询发送到正确的分片上。作为对比，来看一下查询所有数据的过程：

```
> db.users.find().explain()
{
  "clusteredType" : "ParallelSort",
  "shards" : {
    "localhost:30000" : [
      {
        "cursor" : "BasicCursor",
        "nscanned" : 37393,
        "nscannedObjects" : 37393,
        "n" : 37393,
        "millis" : 38,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
```

```
        "indexOnly" : false,
        "indexBounds" : {

        }
      }
    ],
    "localhost:30001" : [
      {
        "cursor" : "BasicCursor",
        "nscanned" : 31303,
        "nscannedObjects" : 31303,
        "n" : 31303,
        "millis" : 37,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {

        }
      }
    ],
    "localhost:30002" : [
      {
        "cursor" : "BasicCursor",
        "nscanned" : 31304,
        "nscannedObjects" : 31304,
        "n" : 31304,
        "millis" : 36,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {

        }
      }
    ]
  },
  "n" : 100000,
  "nChunkSkips" : 0,
  "nYields" : 0,
  "nscanned" : 100000,
  "nscannedObjects" : 100000,
  "millisTotal" : 111,
  "millisAvg" : 37,
  "numQueries" : 3,
  "numShards" : 3
}
```

可以看到，这次查询不得不访问所有 3 个分片，查询出所有数据。通常来说，如果没有在查询中使用片键，mongos 就不得不将查询发送到每个分片。

包含片键的查询能够直接被发送到目标分片或者是集群分片的一个子集，这样的查询叫做定向查询（targeted query）。有些查询必须被发送到所有分片，这样的查询叫做分散 – 聚集查询（scatter-gather query）：mongos 将查询分散到所有分片上，然后将各个分片的查询结果聚集起来。

完成这个实验之后，关闭数据集。切换回最初的 shell，按几次 Enter 键以回到命令行。然后运行 `cluster.stop()` 就可以关闭整个集群了。

```
> cluster.stop()
```

如果不确定某个操作的作用，可以使用 `ShardingTest` 快速创建一个本地集群然后做一些尝试。

第 14 章 配置分片

上一章中，我们在一台机器上创建了一个"集群"。本章讲述如何创建一个更实际的集群，以及分片的配置。

- 创建配置服务器、分片、mongos 进程。
- 增加集群容量。
- 数据的存储和分布。

14.1 何时分片

决定何时分片是一个值得权衡的问题。通常不必太早分片，因为分片不仅会增加部署的操作复杂度，还要求做出设计决策，而该决策以后很难再改。另外最好也不要在系统运行太久之后再分片，因为在一个过载的系统上不停机进行分片是非常困难的。

通常，分片用来：

- 增加可用 RAM；
- 增加可用磁盘空间；
- 减轻单台服务器的负载；
- 处理单个 mongod 无法承受的吞吐量。

因此，良好的监控对于决定应何时分片是十分重要的，必须认真对待其中每一项。由于人们往往过于关注改进其中一个指标，所以应弄明白到底哪一项指标对自己的部署最为重要，并提前做好何时分片以及如何分片的计划。

随着不断增加分片数量，系统性能大致会呈线性增长。但是，如果从一个未分片的系统转换为只有几个分片的系统，性能通常会有所下降。由于迁移数据、维护元数据、路由等开销，少量分片的系统与未分片的系统相比，通常延迟更大，吞吐量甚至可能会更小。因此，至少应该创建 3 个或以上的分片。

14.2 启动服务器

创建集群的第一步是启动所有所需进程。如上章所述，需建立 mongos 和分片。第三个组件——配置服务器也非常重要。配置服务器是普通的 mongod 服务器，保存着集群的配置信息：集群中有哪些分片、分片的是哪些集合，以及数据块的分布。

14.2.1 配置服务器

配置服务器相当于集群的大脑，保存着集群和分片的元数据，即各分片包含哪些数据的信息。因此，应该首先建立配置服务器，鉴于它所包含数据的*极端*重要性，必须启用其日志功能，并确保其数据保存在非易失性驱动器上。每个配置服务器都应位于单独的物理机器上，最好是分布在不同地理位置的机器上。

因 mongos 需从配置服务器获取配置信息，因此配置服务器应先于任何 mongos 进程启动。配置服务器是独立的 mongod 进程，所以可以像启动“普通的”mongod 进程一样启动配置服务器：

```
$ # server-conǸg-1
$ mongod --conǸgsvr --dbpath /var/lib/mongodb -f /var/lib/conǸg/mongod.conf
$
$ # server-conǸg-2
$ mongod --conǸgsvr --dbpath /var/lib/mongodb -f /var/lib/conǸg/mongod.conf
$
$ # server-conǸg-3
$ mongod --conǸgsvr --dbpath /var/lib/mongodb -f /var/lib/conǸg/mongod.conf
```

启动配置服务器时，*不要*使用 `--replSet` 选项：配置服务器不是副本集成员。mongos 会向所有 3 台配置服务器发送写请求，执行一个两步提交类型的操作，以确保 3 台服务器拥有相同的数据，所以这 3 台配置服务器都必须是可写的（在副本集中，只有主节点可以处理客户端的写请求）。



一个常见的疑问是，为什么要用 3 台配置服务器？因为我们需要考虑不时之需。但是，也不需要过多的配置服务器，因为配置服务器上的确认动作是比较耗时的。另外，如果有服务器宕机了，集群元数据就会变成只读的。因此，3 台就足够了，既可以应对不时之需，又无需承受服务器过多带来的缺点。这个数字未来可能会发生变化。

--configsvr 选项指定 mongod 为新的配置服务器。该选项并非必选项，因为它所做的不过是将 mongod 的默认监听端口改为 27019，并把默认的数据目录改为 /data/configdb 而已（可使用 --port 和 --dbpath 选项修改这两项配置）。

但建议使用 --configsvr 选项，因为它比较直白地说明了这些配置服务器的用途。当然，如果不用它启动配置服务器也没问题。

配置服务器并不需要太多的空间和资源。配置服务器的 1 KB 空间约等于 200 MB 真实数据，它保存的只是数据的分布表。由于配置服务器并不需要太多的资源，因此可将其部署在运行着其他程序的机器上，如应用服务器、分片的 mongod 服务器，或 mongos 进程的服务器上。

如果所有的配置服务器都不可用，就要对所有分片做数据分析，以便知道每个分片保存的是什么样的数据。这是可行的，但速度较慢，且令人厌烦。比较好的方式是经常对配置服务器做数据备份。应常在执行集群维护操作之前备份配置服务器的数据。

14.2.2　mongos进程

三个配置服务器均处于运行状态后，启动一个 mongos 进程供应用程序连接。mongos 进程需知道配置服务器的地址，所以必须使用 --conǸgdb 选项启动 mongos：

```
$ mongos --configdb config-1:27019,config-2:27019,config-3:27019 \
> -f /var/lib/mongos.conf
```

默认情况下，mongos 运行在 27017 端口。注意，并不需要指定数据目录（mongos 自身并不保存数据，它会在启动时从配置服务器加载集群数据）。确保正确设置了 logpath，以便将 mongos 日志保存到安全的地方。

可启动任意数量的 mongos 进程。通常的设置是每个应用程序服务器使用一个 mongos 进程（与应用服务器运行在同一台机器上）。

每个 mongos 进程必须按照列表顺序，使用相同的配置服务器列表。

14.2.3　将副本集转换为分片

终于可以添加分片了。有两种可能性：已经有了一个副本集，或是从零开始建立集群。下例假设我们已经拥有了一个副本集。如果是从零开始的话，可先初始化一个空的副本集，然后按照本例的步骤进行后续操作。

如已经有一个使用中的副本集，该副本集会成为第一个分片。为了将副本集转换为分片，需告知 mongos 副本集名称和副本集成员列表。

例如，如果在 server-1、server-2、server-3、server-4、server-5 上有一个名为 spock 的副本集，可连接到 mongos 并运行：

```
> sh.addShard("spock/server-1:27017,server-2:27017,server-4:27017")
{
    "added" : "spock/server-1:27017,server-2:27017,server-4:27017",
    "ok" : true
}
```

可在参数中指定副本集的所有成员，但并非一定要这样做。mongos 能够自动检测到没有包含在副本集成员表中的成员。如运行 `sh.status()`，可发现 MongoDB 已经找到了其他的副本集成员：`"spock/server- 1:27017,server-2:27017,server-4:27017,server-3:27017,server-5:27017"`。

副本集名称 spock 被用作分片名称。如之后希望移除这个分片或是向这个分片迁移数据，可使用 `spock` 来标识这个分片。这比使用特定的服务器名称（如 server-1）要好，因为副本集成员和状态是不断改变的。

将副本集作为分片添加到集群后，就可以将应用程序设置从连接到副本集改为连接到 mongos。添加分片后，mongos 会将副本集内的所有数据库注册为分片的数据库，因此所有查询都会被发送到新的分片上。与客户端库一样，mongos 会自动处理应用故障，将错误返回给客户端。

在开发环境中可测试一下让分片的主节点挂掉，以确保应用程序能够正确处理 mongos 返回的错误。（错误应与直接对话主节点返回的错误相同。）



添加分片后，必须将客户端设置为将所有请求发送到 mongos，而不是副本集。如果客户端仍然把请求直接发送给副本集（而不是通过 mongos）的话，分片是无法正常工作的。添加分片后，应立即将客户端配置为把请求发送给 mongos，同时配置防火墙规则，以确保客户端不能直接将请求发送给分片。

有一个 `--shardsvr` 选项，与前面介绍过的 `--configsvr` 选项类似，它也没什么实用性（只是将默认端口改为 27018），但建议在操作中选择该选项。

也可以创建单 mongod 服务器的分片（而不是副本集分片），但不建议在生产中使用（上一章中的 `ShardingTest` 是这么做的）。直接在 `addShard()` 中指定单个 mongod 的主机名和端口，就可以将其添加为分片了：

```
> sh.addShard("some-server:27017")
```

单一服务器分片默认会被命名为 shard0000、shard0001，依次类推。如打算以后切换为副本集，应先创建一个单成员副本集再添加为分片，而不是直接将单一服务器添加为分片。将单一服务器分片转换为副本集需停机操作（详见 16.3 节）。

14.2.4　增加集群容量

可通过增加分片来增加集群容量。为添加一个新的、空的分片，可先创建一个副本集。确保副本集的名字与其他分片不同。副本集完成初始化并拥有一个主节点后，可在 mongos 上运行 `addShard()` 命令，将副本集作为分片添加到集群中，在参数中指定副本集的名称和主机名作为种子。

如有多个现存的副本集没有作为分片，只要它们没有同名的数据库，就可将它们作为新分片全部添加到集群中。例如，如有一个 blog 数据库的副本集、一个 calendar 数据库的副本集，以及一个 mail、tel、music 数据库的副本集，可将每个副本集作为一个分片添加到集群中，这样就可以得到一个拥有三个分片、五个数据库的集群。但是，如果还有一个数据库名称为 tel 的副本集，那么 mongos 会拒绝将这个副本集作为分片添加到集群中。

14.2.5　数据分片

除非明确指定规则，否则 MongoDB 不会自动对数据进行拆分。如有必要，必须明确告知数据库和集合。

假设我们希望对 music 数据库中的 artists 集合按照 `name` 键进行分片。首先，对 music 数据库启用分片：

```
> db.enableSharding("music")
```

对数据库分片是对集合分片的先决条件。

对数据库启用分片后，就可以使用 `shardCollection()` 命令对集合分片了：

```
> sh.shardCollection("music.artists", {"name" : 1})
```

现在，集合会按照 `name` 键进行分片。如果是对已存在的集合进行分片，那么 `name` 键上必须有索引，否则 `shardCollection()` 会返回错误。如果出现了错误，就先创建索引（mongos 会建议创建的索引作为错误消息的一部分返回），然后重试 `shardCollection()` 命令。

如要进行分片的集合还不存在，mongos 会自动在片键上创建索引。

`shardCollection()` 命令会将集合拆分为多个数据块，这是 MongoDB 迁移数据的基本单元。命令成功执行后，MongoDB 会均衡地将集合数据分散到集群的分片上。这个过程不是瞬间完成的，对于比较大的集合，可能会花费几个小时才能完成。

14.3　MongoDB如何追踪集群数据

每个 mongos 都必须能够根据给定的片键找到文档的存放位置。理论上来说，MongoDB 能够追踪到每个文档的位置，但当集合中包含成百上千万个文档的时候，就会变得难以操作。因此，MongoDB 将文档分组为块（chunk），每个块由给定片键特定范围内的文档组成。一个块只存在于一个分片上，所以 MongoDB 用一个比较小的表就能够维护块跟分片的映射。

例如，如用户集合的片键是 `{"age" : 1}`，其中某个块可能是由 `age` 值为 3~17 的文档组成的。如果 mongos 得到一个 `{"age" : 5}` 的查询请求，它就可以将查询路由到 `age` 值为 3~17 的块所在的分片。

进行写操作时，块内的文档数量和大小可能会发生改变。插入文档可使块包含更多的文档，删除文档则会减少块内文档的数量。如果我们针对儿童和中小学生制作游戏，那么这个 `age` 值为 3~17 的块可能会变得越来越大。几乎所有的用户都会被包含在这个块内，且在同一分片上。这就违背了我们分布式存放数据的初衷。因此，当一个块增长到特定大小时，MongoDB 会自动将其拆分为两个较小的块。在本例中，该块可能会被拆分为一个 `age` 值为 3~11 的块和一个 `age` 值为 12~17 的块。注意，这两个小块包含了之前大块的所有文档以及 `age` 的全部域值。这些小块变大后，会被继续拆分为更小的块，直到包含 `age` 的全部域值。

块与块之间的 `age` 值范围不能有交集，如 3~15 和 12~17。如果存在交集的话，那么 MongoDB 为了查询处于交集中的 `age` 值（如 14）时，则需分别查找这两个块。只在一个块中进行查找效率会更高，尤其是在块分散在集群中时。

一个文档，属于且只属于一个块。这意味着，不可以使用数组字段作为片键，因为 MongoDB 会为数组创建多个索引条目。例如，如某个文档的 `age` 字段值是 `[5, 26, 83]`，该文档就会出现在三个不同的块中。



一个常见的误解是同一个块内的数据保存在磁盘的同一片区域。这是不正确的，块并不影响 mongod 保存集合数据的方式。

14.3.1 块范围

可使用块包含的文档范围来描述块。新分片的集合起初只有一个块，所有文档都位于这个块中。此块的范围是负无穷到正无穷，在 shell 中用 `$minKey` 和 `$maxKey` 表示。

随着块的增长，MongoDB 会自动将其分成两个块，范围分别是负无穷到 <*some value*> 和 <*some value*> 到正无穷。两个块中的 <*some value*> 值相同，范围较小的块包含比 <*some value*> 小的所有文档（但不包含 <*some value*> 值），范围较大的块包含从 <*some value*> 一直到正无穷的所有文档（包含 <*some value*> 值）。

用一个例子来更直观地说明：假如我们按照之前提到的 `"age"` 字段进行分片。所有 `"age"` 值为 3~17 的文档都包含在一个块中：$3 \leqslant \texttt{age} < 17$。该块被拆分后，我们得到了两个较小的块，其中一个范围是 $3 \leqslant \texttt{age} < 12$，另一个范围是 $12 \leqslant \texttt{age} < 17$。这里的 12 就叫做拆分点（split point）。

块信息保存在 config.chunks 集合中。查看集合内容，会发现其中的文档如下（简洁起见，这里忽略了一些字段）：

```
> db.chunks.find(criteria, {"min" : 1, "max" : 1})
{
    "_id" : "test.users-age_-100.0",
    "min" : {"age" : -100},
    "max" : {"age" : 23}
}
{
    "_id" : "test.users-age_23.0",
    "min" : {"age" : 23},
    "max" : {"age" : 100}
}
{
    "_id" : "test.users-age_100.0",
    "min" : {"age" : 100},
    "max" : {"age" : 1000}
 }
```

基于以上 config.chunks 文档，不同文档在块中的分布情况如下例所示：

- `{"_id" : 123, "age" : 50}`
  该文档位于第二个块中，因为第二个块包含 `age` 值为 23~100 的所有文档。

- `{"_id" : 456, "age" : 100}`
  该文档位于第三个块中，因为较小的边界值是包含在块中的。第二个块包含了 `age` 值小于 100 的所有文档，但不包含等于 100 的文档。

- `{"_id" : 789, "age" : -101}`
  该文档不位于上面所示的这些块中，而是位于一个比第一个块范围更小的块中。

可使用复合片键，工作方式与使用复合索引进行排序一样。假如在 `{"username" : 1, "age" : 1}` 上有一个片键，那么可能会存在如下块范围：

```
{
    "_id" : "test.users-username_MinKeyage_MinKey",
    "min" : {
        "username" : { "$minKey" : 1 },
        "age" : { "$minKey" : 1 }
    },
    "max" : {
        "username" : "user107487",
        "age" : 73
    }
}
{
    "_id" : "test.users-username_\"user107487\"age_73.0",
    "min" : {
        "username" : "user107487",
        "age" : 73
    },
    "max" : {
        "username" : "user114978",
        "age" : 119
    }
}
{
    "_id" : "test.users-username_\"user114978\"age_119.0",
    "min" : {
        "username" : "user114978",
        "age" : 119
    },
    "max" : {
        "username" : "user122468",
        "age" : 68
    }
}
```

因此，对于一个给定的用户名（或者是用户名和年龄），mongos 可轻易找到其所对应的文档。但如果只给定年龄，mongos 就必须查看所有（或者几乎所有）块。如果希望基于 `age` 的查询能够被路由到正确的块上，则需使用“相反”的片键：`{"age" : 1, "username" : 1}`。从这个例子中我们可以得出一个结论：基于片键第二个字段的范围可能会出现在多个块中。

14.3.2 拆分块

mongos 会记录在每个块中插入了多少数据，一旦达到某个阈值，就会检查是否需要对块进行拆分，如图 14-1 和图 14-2 所示。如果块确实需要被拆分，mongos 就会在

配置服务器上更新这个块的元信息。块拆分只需改变块的元数据即可，而无需进行数据移动。进行拆分时，配置服务器会创建新的块文档，同时修改旧的块范围（即 "max" 值）。拆分完成后，mongos 会重置对原始块的追踪器，同时为新的块创建新的追踪器。

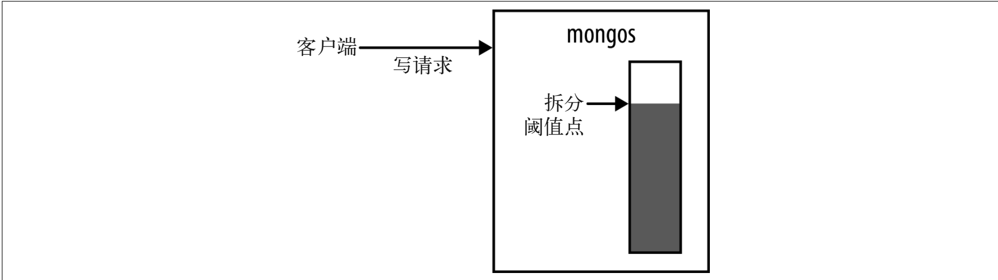


图 14-1：收到客户端发起的写请求时，mongos 会检查当前块的拆分阈值点

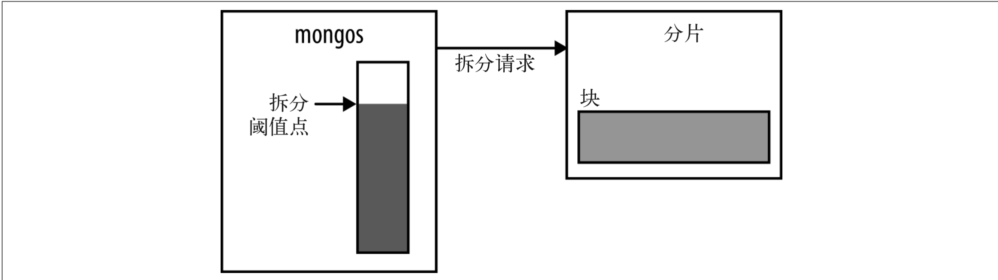


图 14-2：如果达到了拆分阈值点，mongos 就会向分片发起一个针对该拆分点的拆分请求

mongos 向分片询问某块是否需被拆分时，分片会对块大小进行粗略的计算。如果发现块正在不断变大，它就会计算出合适的拆分点，然后将这些信息发送给 mongos，如图 14-3 所示。

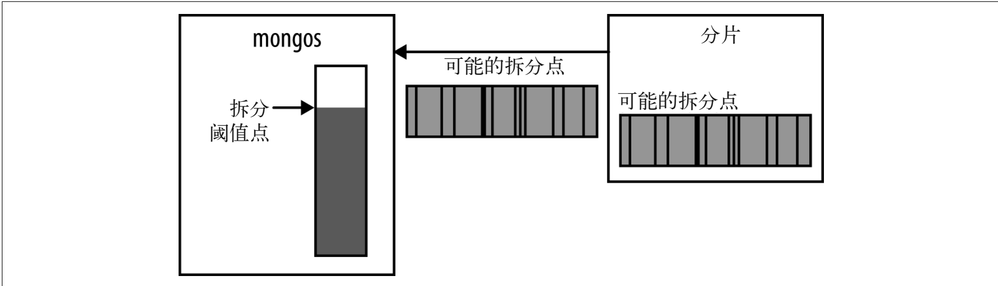


图 14-3：分片计算块的拆分点，并将这些信息发回 mongos

分片有时可能会找不到任何可用的拆分点（即使此块较大），因为合法拆分块方法有限。具有相同片键的文档必须保存在相同的块中，因此块只能在片键的值发生变化

的点对块进行拆分。例如，如果片键的值等于 age 的值，则下列块可在片键发生变化的点被拆分：

```
{"age" : 13, "username" : "ian"}
{"age" : 13, "username" : "randolph"}
------------ // 拆分点
{"age" : 14, "username" : "randolph"}
{"age" : 14, "username" : "eric"}
{"age" : 14, "username" : "hari"}
{"age" : 14, "username" : "mathias"}
------------ // 拆分点
{"age" : 15, "username" : "greg"}
{"age" : 15, "username" : "andrew"}
```

mongos 无需在每个可用的拆分点对块进行拆分，但拆分时只能从这些拆分点中选择一个。

例如，如果块包含下列文档，则此块不可拆分，除非应用开始插入不同片键的文档：

```
{"age" : 12, "username" : "kevin"}
{"age" : 12, "username" : "spencer"}
{"age" : 12, "username" : "alberto"}
{"age" : 12, "username" : "tad"}
```

因此，拥有不同的片键值是非常重要的。其他重要属性会在下一章讲到。

如果在 mongos 试图进行拆分时有一个配置服务器挂了，那么 mongos 就无法更新元数据，如图 14-4 所示。在进行拆分时，所有配置服务器都必须可用且可达。mongos 如果不断接收到块的写请求，则会处于尝试拆分与拆分失败的循环中。只要配置服务器不可用于拆分，拆分就无法进行，mongos 不断发起的拆分请求就会拖慢 mongos 和当前分片（每次收到的写请求都会重复图 14-1 到图 14-4 演示的过程）。这种 mongos 不断重复发起拆分请求却无法进行拆分的过程，叫做拆分风暴（split storm）。防止拆分风暴的唯一方法是尽可能保证配置服务器的可用和健康。也可重启 mongos，重置写入计数器，这样它就不再处于拆分阈值点了。

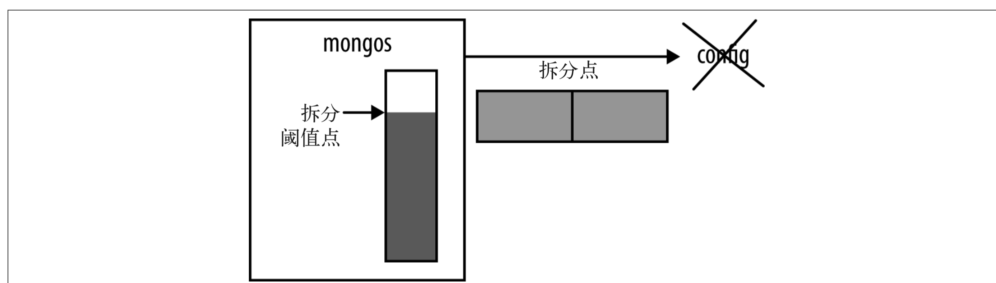


图 14-4：mongos 选择一个拆分点，然后试图将这些信息通知给配置服务器，但是配置服务器不可达。因此，它仍位于这个块的拆分阈值点。随后的任何写请求都会重复上面的过程

另一个问题是，mongos 可能不会意识到它需要拆分一个较大的块。并没有一个全局的计数器用于追踪每个块到底有多大。每个 mongos 只是计算其收到的写请求是否达到了特定的阈值点（如图 14-5 所示）。也就是说，如果 mongos 进程频繁地上线和宕机，那么 mongos 在再次宕机之前可能永远无法收到足以达到拆分阈值点的写请求，因此块会变得越来越大，如图 14-6 所示。

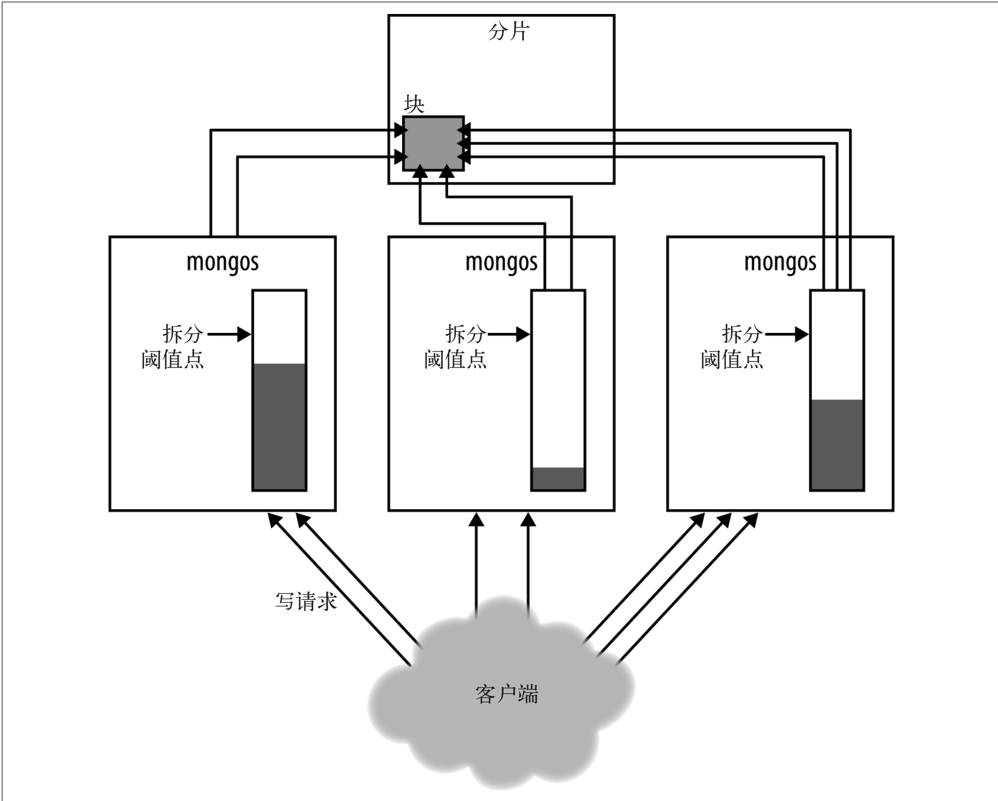


图 14-5：随着 mongos 进程不断执行写请求，它们的计数器也会不断增长，直至拆分阈值点

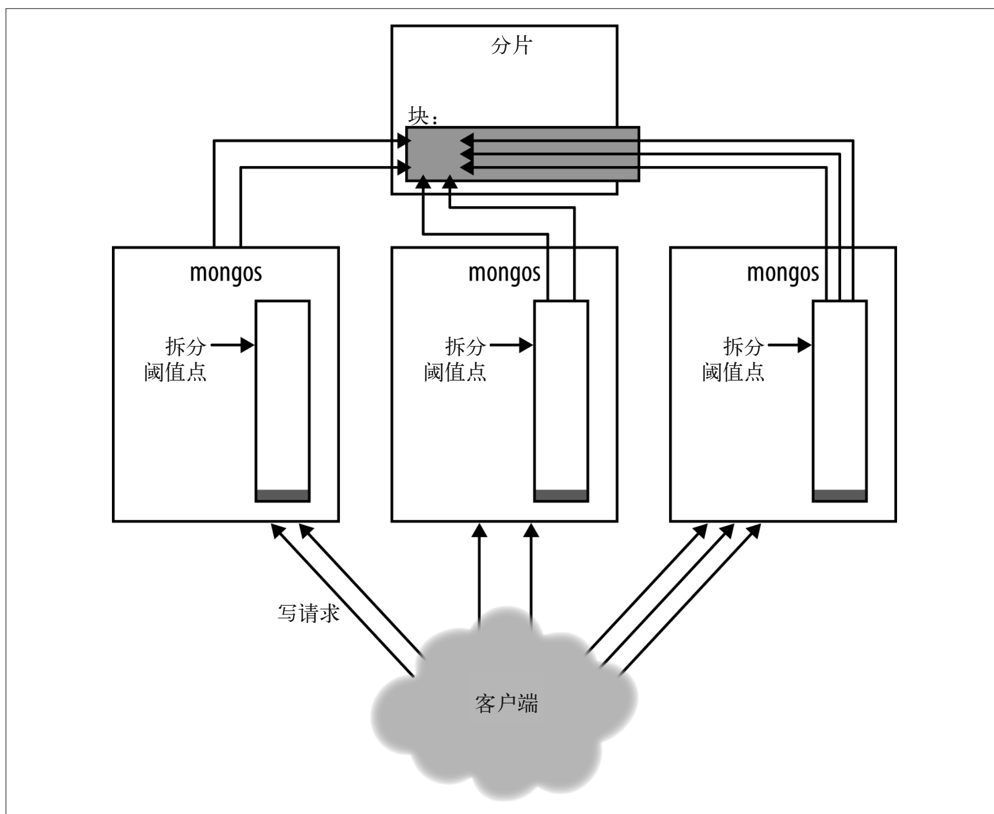


图 14-6：如果 mongos 进程不断重启，它们的计数器可能永远也不会到达阈值点，因此块的增长不存在最大值

防止这种情况发生的第一种方式是减少 mongos 进程的波动。尽可能保证 mongos 进程可用，而不是在需要的时候将其开启，不需要的时候又将其关掉。然而，实际部署中可能会发现，维持不需要的 mongos 持续运行开销过大。这时可选用另一种方式：使块的大小比实际预期稍小些，这样就更容易达到拆分阈值点。

可在启动 mongos 时指定 `--nosplit` 选项，从而关闭块的拆分。

14.4 均衡器

均衡器（balancer）负责数据的迁移。它会周期性地检查分片间是否存在不均衡，如果存在，则会开始块的迁移。虽然均衡器通常被看作是单一实体，但每个 mongos 有时也会扮演均衡器的角色。

每隔几秒钟，mongos 就会尝试变身为均衡器。如果没有其他可用的均衡器，mongos 就会对整个集群加锁，以防止配置服务器对集群进行修改，然后做一次均衡。均衡并不会影响 mongos 的正常路由操作，所以使用 mongos 的客户端不会受到影响。

查看 config.locks 集合，可得知哪一个 mongos 是均衡器：

```
> db.locks.findOne({"_id" : "balancer"})
{
    "_id" : "balancer",
    "process" : "router-23:27017:1355763351:1804289383",
    "state" : 0,
    "ts" : ObjectId("50cf939c051fcdb8139fc72c"),
    "when" : ISODate("2012-12-17T21:50:20.023Z"),
    "who" : "router-23:27017:1355763351:1804289383:Balancer:846930886",
    "why" : "doing balance round"
}
```

config.locks 集合会追踪所有集群范围的锁。`_id` 为 `balancer` 的文档就是均衡器。从其中的 `who` 字段可得知当前或曾经作为均衡器的 mongos 是哪一个：在本例中是 router-23:27017。`state` 字段表明均衡器是否正在运行：0 表示处于非活动状态，2 表示正在进行均衡（1 表示 mongos 正在尝试得到锁，但还没有得到，通常不会看到状态 1）.

mongos 成为均衡器后，就会检查每个集合的分块表，从而查看是否有分片达到了均衡阈值（balancing threshold）。不均衡的表现指，一个分片明显比其他分片拥有更多的块（精确的阈值有多种不同情况：集合越大越能承受不均衡状态）。如果检测到不均衡，均衡器就会开始对块进行再分布，以使每个分片拥有数量相当的块。如果没有集合达到均衡阈值，mongos 就不再充当均衡器的角色了。

假如有一些集合到达了阈值，均衡器则会开始做块迁移。它会从负载比较大的分片中选择一个块，并询问该分片是否需要在迁移之前对块进行拆分。完成必要的拆分后，就会将块迁移至块数量较少的机器上。

使用集群的应用程序无需知道数据迁移：在数据迁移完成之前，所有的读写请求都会被路由到旧的块上。如果元数据更新完成，那么所有试图访问旧位置数据的 mongos 进程都会得到一个错误。这些错误应该对客户端不可见：mongos 会对这些错误做静默处理，然后在新的分片上重新执行之前的操作。

有时会在 mongos 的日志中看到“unable to `setShardVersion`”的信息，这是一种很常见的错误。mongos 在收到这种错误时，会查看配置服务器数据的新位置，并更新块分布表，然后重新执行之前的请求。如果成功从新的位置得到了数据，则会将

数据返回给客户端。除了日志中会记录一条错误日志外，整个过程好像什么错误都没有发生过一样。

如果由于配置服务器不可用导致 mongos 无法获取块的新位置，则会向客户端返回错误。所以，应尽可能保证配置服务器处于可用状态。

第 15 章

选择片键

使用分片时，最重要也是最困难的任务就是选择数据的分发方式。需要理解 MongoDB 的数据分发机制才能够做出明智的选择。本章旨在帮助大家更好地选择片键，内容包括：

- 如何在多个可用的片键中做出选择；
- 不同使用场景中的片键选择；
- 哪些键不能作为片键；
- 自定义数据分发方式的可选策略；
- 如何手动对数据分片。

由于前几章已经讲述了分片的基本知识，所以本章假设大家对分片已有基本的了解。

15.1 检查使用情况

对集合进行分片时，要选择一或两个字段用于拆分数据。这个键（或这些键）就叫做*片键*。一旦拥有多个分片，再修改片键几乎是不可能的事情，因此选择合适的片键（或者至少快速注意到可能存在的问题）是非常重要的。

为了选择合适的片键，需了解自己的工作量以及片键是如何对应用程序的请求进行分发的。这个问题不太好描述，可以尝试一些小例子，或者是在备用数据集上做一些实验。本节含有大量图表和解释说明，但最好的方式还是在自己的数据集上试一试。

对集合进行分片前，先回答以下问题。

- 计划做多少个分片？拥有 3 个分片的集群比拥有 1000 个分片的集群更具有灵活性。随着集群变得越来越大，不应做那些需要查询所有分片的查询，因此几乎所有查询都须包含片键。
- 分片是为了减少读写延迟吗？（*延迟*指某个操作花费的时间，如写操作花费 20 毫秒，但我们需要将其缩减至 10 毫秒）。降低写延迟的方式通常是将请求发送到地理位置更近的服务器或者是更强大的机器上。
- 分片是为了增加读写吞吐量吗？（*吞吐量*指集群在同一时间能够处理的请求数量：集群能够在 20 毫秒内处理 1000 次写请求，但我们需要其能够在 20 毫秒内处理 5000 次写请求）。增加吞吐量通常需要提高并行性，并确保请求被均衡地分发到各集群成员上。
- 分片是为了增加系统资源吗？（比如，每 GB 数据提供 MongoDB 更多的可用 RAM）。如果是这样，可能会希望尽量保持工作集较小。

根据这些问题来对不同片键进行评估，并判断所选片键是否适用于自己的情况。这样做能够提供所需的目标查询吗？能够按所需方式提高系统吞吐量或者减少读写延迟吗？如需保持工作集的小巧，这样做可以达到要求吗？

15.2　数据分发

拆分数据最常用的数据分发方式有三种：升序片键（ascending key）、随机分发的片键和基于位置（location-based）的片键。也有一些其他类型的键可供使用，但大部分都属于这三种类别。以下几节会分别介绍这三种方式。

15.2.1　升序片键

升序片键通常有点类似于 `"date"` 字段或者是 `ObjectId`，是一种会随着时间稳定增长的字段。自增长的主键是升序键的另一个例子，但它很少出现在 MongoDB 中，除非要从其他数据库中导入数据。

假设我们依据升序键做分片，如使用 `ObjectId` 的集合中的 `"_id"` 键。如果基于 `"_id"` 分片，那么集合就会依据不同的 `"_id"` 范围被拆分为多个块，如图 15-1 所示。这些块会分发在我们这个拥有分片的集群中（比如说 3 个分片），如图 15-2 所示。

假设要创建一个新文档，它会位于哪个块呢？答案是范围为 `ObjectId("5112fae0b4a4b396ff9d0ee5")` 到 `$maxKey` 的块。这个块叫做*最大块*（max chunk），因为该块包含有 `$maxKey`。

$minKey -> ObjectId("5112fa61b4a4b396ff960262")

ObjectId("5112fa61b4a4b396ff960262") -> ObjectId("5112fa9bb4a4b396ff96671b")

ObjectId("5112fa9bb4a4b396ff96671b") -> ObjectId("5112faa0b4a4b396ff9732db")

ObjectId("5112faa0b4a4b396ff9732db") -> ObjectId("5112fabbb4a4b396ff97fb40")

ObjectId("5112fabbb4a4b396ff97fb40") -> ObjectId("5112fac0b4a4b396ff98c6f8")

ObjectId("5112fac0b4a4b396ff98c6f8") -> ObjectId("5112fac5b4a4b396ff998b59")

ObjectId("5112fac5b4a4b396ff998b59") -> ObjectId("5112facab4a4b396ff9a56c5")

ObjectId("5112facab4a4b396ff9a56c5") -> ObjectId("5112facfb4a4b396ff9b1b55")

ObjectId("5112facfb4a4b396ff9b1b55") -> ObjectId("5112fad4b4a4b396ff9bd69b")

ObjectId("5112fad4b4a4b396ff9bd69b") -> ObjectId("5112fae0b4a4b396ff9d0ee5")

ObjectId("5112fae0b4a4b396ff9d0ee5") -> $maxKey

图 15-1：集合依据不同的 `ObjectId` 范围被拆分，每个范围都是一个块

如果再插入一个文档，它也会出现在最大块中。事实上，接下来的每个新文档都会被插入到最大块中！每一个插入文档的 `"_id"` 字段值都会比之前文档的 `"_id"` 字段值更接近正无穷（因为 `ObjectId` 一直在增长），所以这些文档都会插入到最大块中。

这样会带来一些有趣的属性，通常都是些不良属性。首先，所有的写请求都会被路由到一个分片（本例中是 shard0002）中。该块是唯一一个不断增长和拆分的块，因为它是唯一一个能够接收到插入请求的块。随着新数据的不断插入，该最大块会不断拆分出新的小块，如图 15-3 所示。

这种模式经常会导致 MongoDB 的数据均衡处理变得更为困难，因为所有的新块都是由同一分片创建的。因此，MongoDB 必须不断将一些块移至其他分片，而不能像在一个比较均衡分发的系统中那样，只需纠正那些比较小的不均衡就好了。

分片 0000

ObjectId("5112fa9bb4a4b396ff96671b") -> ObjectId("5112faa0b4a4b396ff9732db")

ObjectId("5112faa0b4a4b396ff9732db") -> ObjectId("5112fabbb4a4b396ff97fb40")

ObjectId("5112fabbb4a4b396ff97fb40") -> ObjectId("5112fac0b4a4b396ff98c6f8")

分片 0001

$minKey -> ObjectId("5112fa61b4a4b396ff960262")

ObjectId("5112fa61b4a4b396ff960262") -> ObjectId("5112fa9bb4a4b396ff96671b")

ObjectId("5112fac0b4a4b396ff98c6f8") -> ObjectId("5112fac5b4a4b396ff998b59")

ObjectId("5112fac5b4a4b396ff998b59") -> ObjectId("5112facab4a4b396ff9a56c5")

分片 0002

ObjectId("5112facab4a4b396ff9a56c5") -> ObjectId("5112facfb4a4b396ff9b1b55")

ObjectId("5112facfb4a4b396ff9b1b55") -> ObjectId("5112fad4b4a4b396ff9bd69b")

ObjectId("5112fad4b4a4b396ff9bd69b") -> ObjectId("5112fae0b4a4b396ff9d0ee5")

ObjectId("5112fae0b4a4b396ff9d0ee5") -> $maxKey

图 15-2：块在分片中是以随机顺序分发的

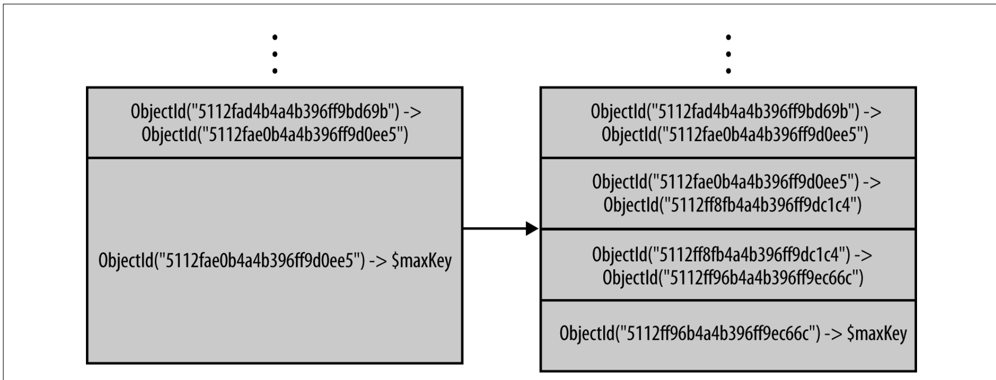


图 15-3：最大块不断增长，不断被拆分为多个块

15.2.2 随机分发的片键

另一种方式是随机分发的片键。随机分发的键可以是用户名、邮件地址、UDID（Unique Device IDentifier，唯一设备标识符 ）、MD5 散列值，或者是数据集中其他一些没有规律的键。

假如片键是 0 和 1 之间的随机数，各分片上随机分发的块如图 15-4 所示。

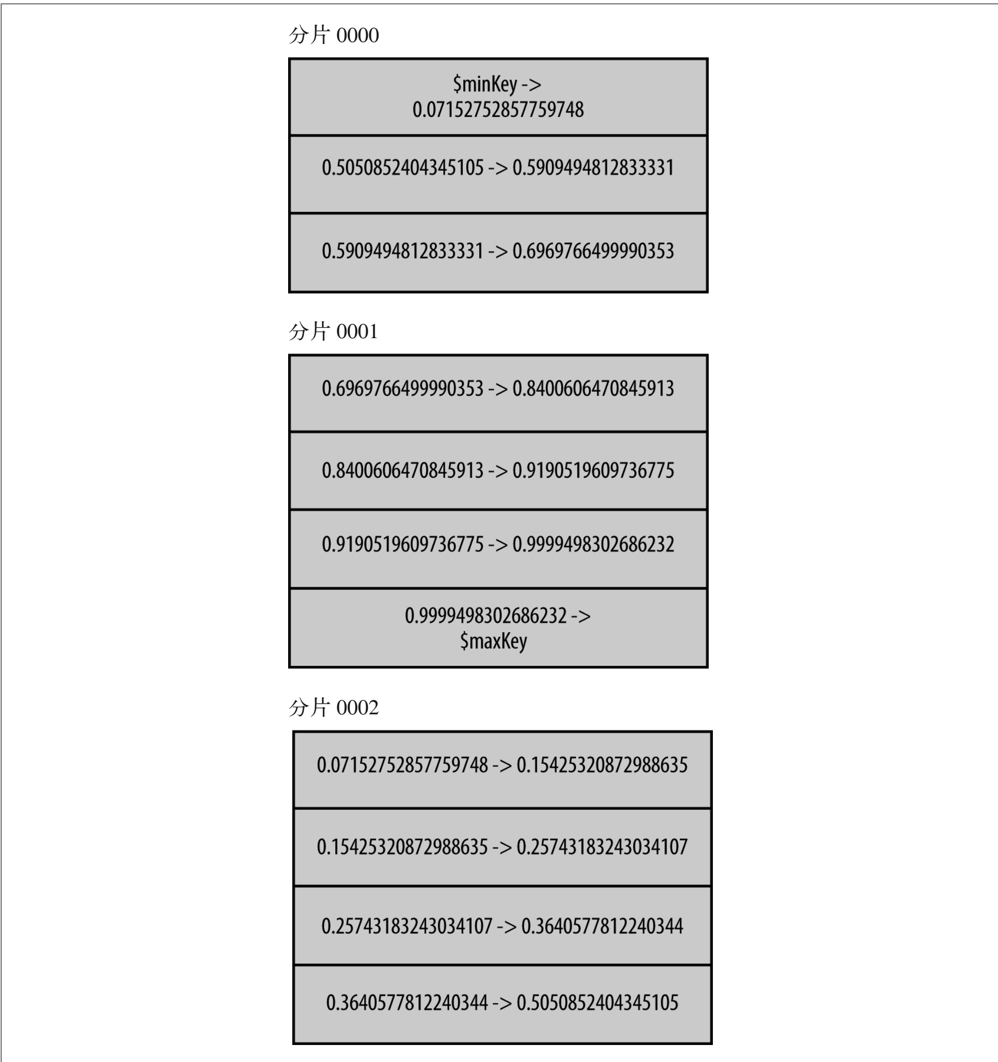


图 15-4：如前一节所述，块随机地分发在集群中

随着更多的数据被插入，数据的随机性意味着，新插入的数据会比较均衡地分发在不同的块中。可以试着插入 10 000 个文档，来验证一下会发生什么：

```
> var servers = {}
> var findShard = function (id) {
...     var explain = db.random.find({_id:id}).explain();
...     for (var i in explain.shards) {
...         var server = explain.shards[i][0];
...         if (server.n == 1) {
...             if (server.server in servers) {
...                 servers[server.server]++;
...             } else {
...                 servers[server.server] = 1;
...             }
...         }
...      }
... }
> for (var i = 0; i < 10000; i++) {
...     var id = ObjectId();
...     db.random.insert({"_id" : id, "x" : Math.random()});
...     findShard(id);
... }
> servers
{
    "spock:30001" : 2942,
    "spock:30002" : 4332,
    "spock:30000" : 2726
}
```

由于写入数据是随机分发的，各分片增长的速度应大致相同，这就减少了需要进行迁移的次数。

使用随机分发片键的唯一弊端在于，MongoDB 在随机访问超出 RAM 大小的数据时效率不高。然而，如果拥有足够多的 RAM 或者是并不介意系统性能的话，使用随机片键在集群上分配负载是非常好的。

15.2.3 基于位置的片键

基于位置的片键可以是用户的 IP、经纬度，或者是地址。位置片键不必与实际的物理位置字段相关：这里的“位置”比较抽象，数据会依据这个“位置”进行分组。无论如何，所有与该键值比较接近的文档都会被保存在同一范围的块中。这样可以比较方便地将数据与相应的用户，以及相关联的数据保存在一起。

例如，假设我们有一个集合的文档是按照 IP 地址进行分片的。文档会依据 IP 地址被分成不同的块，并随机分布在集群中，如图 15-5 所示。

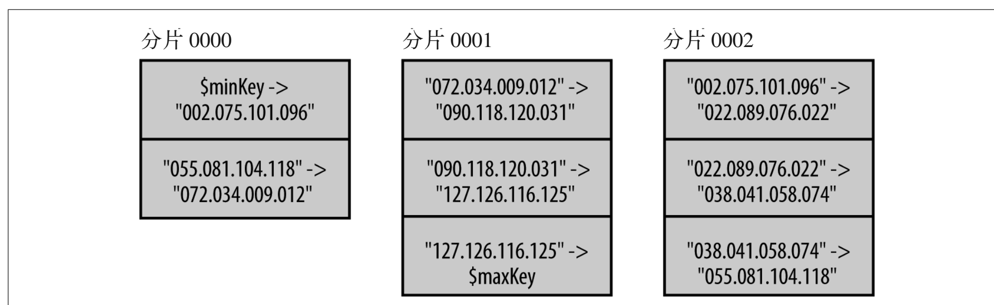


图 15-5：IP 地址集合中的块分发情况

如果希望特定范围的块出现在特定的分片中，可以为分片添加 `tag`，然后为块指定相应的 `tag`。在本例中，假如我们希望特定范围的 IP 段出现在特定的分片中，比如让“56.*.*.*”（美国邮政署的 IP 段）出现在 shard0000，让“17.*.*.*”（苹果公司的 IP 段）出现在 shard0000 或 shard0002 上。我们并不关心其他的 IP 出现在什么位置。可通过为分片指定 `tag`，请求均衡器实现该指令：

```
> sh.addShardTag("shard0000", "USPS")
> sh.addShardTag("shard0000", "Apple")
> sh.addShardTag("shard0002", "Apple")
```

然后，创建下列规则：

```
> sh.addTagRange("test.ips", {"ip" : "056.000.000.000"},
... {"ip" : "057.000.000.000"}, "USPS")
```

这样就会将所有 IP 地址大于等于 56.0.0.0 和小于 57.0.0.0 的文档分发到标签为“USPS”的分片上。接下来，再为苹果公司的 IP 段添加一条规则：

```
> sh.addTagRange("test.ips", {"ip" : "017.000.000.000"},
... {"ip" : "018.000.000.000"}, "Apple")
```

均衡器在移动块时，会试图将这些范围的块移动到这些分片上。注意，该过程不会立即生效。没有被打过标签的块仍会正常移动。均衡器会继续尝试将块均衡地分发在不同的分片上。

15.3 片键策略

本节我们将学习针对不同类型应用程序的几种片键选项。

15.3.1 散列片键

如果追求的是数据加载速度的极致，那么散列片键（Hashed Shard Key）是最佳选择。散列片键可使其他任何键随机分发，因此，如果打算在大量查询中使用升序键，但同时又希望写入数据随机分发的话，散列片键会是个非常好的选择。

弊端是无法使用散列片键做指定目标的范围查询。如无需做范围查询，那么散列片键就非常合适。

创建一个散列片键，首先要创建散列索引：

```
> db.users.ensureIndex({"username" : "hashed"})
```

然后对集合分片：

```
> sh.shardCollection("app.users", {"username" : "hashed"})
{ "collectionsharded" : "app.users", "ok" : 1 }
```

如果在一个不存在的集合上创建散列片键，`shardCollection` 的行为会比较有趣：它假设我们希望对数据块进行均衡分发，所以会立即创建一些空的块，并将这些块分发在集群中。例如，在创建散列片键之前，集合如下：

```
> sh.status()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
        {  "_id" : "shard0000", "host" : "localhost:30000" }
        {  "_id" : "shard0001", "host" : "localhost:30001" }
        {  "_id" : "shard0002", "host" : "localhost:30002" }
  databases:
        {  "_id" : "admin", "partitioned" : false, "primary" : "config" }
        {  "_id" : "test", "partitioned" : true, "primary" : "shard0001" }
```

`shardCollection` 命令返回后，每个分片上立即出现了两个块，并均衡地分发在整个集群中：

```
> sh.status()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
    {  "_id" : "shard0000", "host" : "localhost:30000" }
    {  "_id" : "shard0001", "host" : "localhost:30001" }
    {  "_id" : "shard0002", "host" : "localhost:30002" }
  databases:
    {  "_id" : "admin", "partitioned" : false, "primary" : "conǸg" }
    {  "_id" : "test", "partitioned" : true, "primary" : "shard0001" }
        test.foo
            shard key: { "username" : "hashed" }
```

```
chunks:
    shard0000       2
    shard0001       2
    shard0002       2
{ "username" : { "$MinKey" : true } }
    -->> { "username" : NumberLong("-6148914691236517204") }
    on : shard0000 { "t" : 3000, "i" : 2 }
{ "username" : NumberLong("-6148914691236517204") }
    -->> { "username" : NumberLong("-3074457345618258602") }
    on : shard0000 { "t" : 3000, "i" : 3 }
{ "username" : NumberLong("-3074457345618258602") }
    -->> { "username" : NumberLong(0) }
    on : shard0001 { "t" : 3000, "i" : 4 }
{ "username" : NumberLong(0) }
    -->> { "username" : NumberLong("3074457345618258602") }
    on : shard0001 { "t" : 3000, "i" : 5 }
{ "username" : NumberLong("3074457345618258602") }
    -->> { "username" : NumberLong("6148914691236517204") }
    on : shard0002 { "t" : 3000, "i" : 6 }
{ "username" : NumberLong("6148914691236517204") }
    -->> { "username" : { "$MaxKey" : true } }
    on : shard0002 { "t" : 3000, "i" : 7 }
```

注意，现在集合中还没有文档，但当插入新文档时，写请求一开始就会被均衡地分发到不同的分片上。通常需要等待块的增长与拆分，直到块移动时再将写请求分发到其他分片上。使用这种自动机制，数据块从一开始就会均衡地分发在所有分片上。

使用散列片键存在着一定的局限性。首先，不能使用 `unique` 选项。其次，与其他片键一样，不能使用数组字段。最后注意，浮点型的值会先被取整，然后才会进行散列，所以 1 和 1.999999 会得到相同的散列值。

15.3.2 GridFS的散列片键

在对 GridFS 集合做分片之前，确保已理解了 GridFS 的数据存储机制（第 6 章有详细介绍）。

在接下来的介绍中，"块"（chunks）这一术语会存在多重含义，因为 GridFS 会将文件拆分为块，而分片也会将集合拆分为块。因此，在本章后续内容中，分别以"GridFS 块"和"分片块"表示这两种块。

GridFS 集合通常来说非常适合做分片，因为它们包含大量的文件数据。但是，在 `fs.chunks` 上自动创建的索引并不是特别适合作为分片键：`{"_id" : 1}` 是一个升序键，`{"files_id" : 1, "n" : 1}` 使用了 `fs.files` 的 `_id` 字段，因此它也是一个升序键。

但是，如果在 `"files_id"` 字段上创建散列索引，则每个文件都会被随机分发到集

群中。但是一个文件只能被包含在一个单一的块中。这是非常好的，因为，写请求被均衡地分发到所有分片上，而读取文件数据时只需查询一个单一的分片即可。

为实现这种策略，必须在 `{"files_id" : "hashed"}` 上创建新的索引（在本书编写之时，`mongos` 还不支持使用复合索引的子集作为片键）。然后依据这个字段对集合分片：

```
> db.fs.chunks.ensureIndex({"files_id" : "hashed"})
> sh.shardCollection("test.fs.chunks", {"files_id" : "hashed"})
{ "collectionsharded" : "test.fs.chunks", "ok" : 1 }
```

另外提醒一下，由于 fs.files 集合比 fs.chunks 集合小得多，fs.files 集合可能需要做分片，也可能不需要。可以对该集合做分片，但通常没什么必要。

15.3.3 流水策略

如果有一些服务器比其他服务器更强大，我们可能会希望让这些强大的服务器处理更多的负载。比如说，假如有一个使用 SSD 的分片能够处理 10 倍于其他机器（使用转式磁盘）的负载。幸运的是，我们有 10 个其他分片。可强制将所有新数据插入到 SSD，然后让均衡器将旧的块移动到其他分片上。这样能够提供比转式磁盘更低的延迟。

为实现这种策略，需将最大范围的块分布在 SSD 上。首先，为 SSD 指定一个标签：

```
> sh.addShardTag("shard-name", "ssd")
```

将升序键的当前值一直到正无穷范围的块指定分布在 SSD 分片上，以便后续的写入请求均被分发到 SSD 分片上：

```
> sh.addTagRange("dbName.collName", {"_id" : ObjectId()},
... {"_id" : MaxKey}, "ssd")
```

现在，所有的插入请求均会被路由到这个块上，这个块始终位于标签为 `ssd` 的分片上。

但是，除非修改标签范围，否则从升序键的当前值一直到正无穷的这个范围则被固定在了这个分片上。可创建一个定时任务每天更新一次标签范围，如下：

```
> use config
> var tag = db.tags.findOne({"ns" : "dbName.collName",
... "max" : {"shardKey" : MaxKey}})
> tag.min.shardKey = ObjectId()
> db.tags.save(tag)
```

这样，前一天的块就可以被移动到其他分片上了。

此策略的另一弊端是需做一些修改才能进行扩展。如果写请求超出了 SSD 的处理能力，想要将负载均衡地分布到当前服务器和另一台服务器并不简单。

如果没有高性能服务器来处理插入流水，或者是没有使用标签，那么不要将升序键用作片键。否则，所有写请求都会被路由到同一分片上。

15.3.4 多热点

单个 mongod 服务器在处理升序写请求时是最有效的。这种技术与分片相冲突，写请求分布在集群中时，分片是最高效的。这种技术会创建多个热点（最好在每个分片都创建几个热点），写请求于是会均衡地分布在集群内，而在单个分片上则是以升序分布的。

为实现这种方式，需使用复合片键（compound shard key）。复合片键中的第一个值只是比较粗略的随机值，势也比较低。可将片键第一部分中的每个值想象为一个块，如图 15-6 所示。随着插入数据的增多，这种现象也会随之出现，虽然可能不会被分离得这么整洁（注意图中的 `$minKey` 行）。但是，如果插入足够多的数据，最终会发现基本上每个随机值都位于一个块中。如果继续插入数据，最终同一个随机值则会对应有多个块，这时候就轮到片键中的第二部分出马了。

```
⋮
{"state" : "KS", "_id" : $minKey} ->
{"state" : "KY", "_id" : $minKey}

{"state" : "KY", "_id" : $minKey} ->
{"state" : "LA", "_id" : $minKey}

{"state" : "LA", "_id" : $minKey} ->
{"state" : "MA", "_id" : $minKey}

{"state" : "MA", "_id" : $minKey} ->
{"state" : "MD", "_id" : $minKey}

{"state" : "MD", "_id" : $minKey} ->
{"state" : "ME", "_id" : $minKey}
⋮
```

图 15-6：块的一个子集。每个块都包含一个状态和一个 _id 范围

片键的第二部分是个升序键。也就是说，在一个块内，值总是增加的，如图 15-7 中的文档样例所示。因此，如果每个分片拥有一个块，会是非常完美的配置：写请求在每个分片内都是升序的，如图 15-8 所示。当然，在多个分片中拥有多个块，每个块拥有多个热点，这种方式并不易于扩展：添加一个新的分片不会获得任何写请求，因为这个分片上没有热点块。因此，我们会希望在每个分片上拥有几个热点块（以提供增长空间）。然后，热点块不能过多。少数的热点块能够保持升序写请求的效率。但是，在一个分片上拥有 1000 个“热点”的话，其实写请求就相当于是完全随机的了。

```
{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f1d") }
{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f1e") }
{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f1f") }
{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f20") }
{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f21") }
{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f22") }
{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f23") }
{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f24") }
{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f25") }
```

图 15-7：插入文档的一个样例。注意，所有的 _id 都是升序的

块：{"state" : "CA", "_id" : $minKey} -> {"state" : "CO", "_id" : $minKey}

{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f1f") }

{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f24") }

{ "state" : "CA", "_id" : ObjectId("511bfb9e17d55c62b2371f25") }

块：{"state" : "MA", "_id" : $minKey} -> {"state" : "ME", "_id" : $minKey}

{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f1d") }

{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f21") }

{ "state" : "MA", "_id" : ObjectId("511bfb9e17d55c62b2371f22") }

块：{"state" : "NY", "_id" : $minKey} -> {"state" : "OH", "_id" : $minKey}

{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f1e") }

{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f20") }

{ "state" : "NY", "_id" : ObjectId("511bfb9e17d55c62b2371f23") }

图 15-8：插入的文档被拆分成了多个块。注意，在每个块内，_id 都是升序的

可将这种配置想象成每个块都是一个升序文档的栈。每个分片上拥有多个栈，每个栈都是不断增长的，直到块被拆分。一旦块被拆分，只有一个新块会成为热点块：其他块实际上会处于一种“死掉”的状态，且不会再继续增长。如果这些栈均衡地分发在分片中，那么写请求也会被均衡地分发到不同的分片上。

15.4　片键规则和指导方针

在选择片键前，应注意一些实际限制。

由于与创建索引键的概念类似，因此决定使用哪个键作分片以及创建片键的方法都与之非常相似。事实上，我们使用的片键可能常常就是使用最频繁的索引（或者是索引的变种）。

15.4.1　片键限制

片键不可以是数组。在拥有数组值的键上执行 `sh.shardCollection()`，则命令不会生效。向片键插入数组值也是不被允许的。

文档一旦插入，其片键值就无法修改了。要修改文档的片键值，必须先删除文档，修改片键的值，然后重新插入。因此，应选择不会被改变的字段，或者是很少发生改变的字段。

大多特殊类型的索引都不能被用作片键。特别是不能在地理空间索引上进行分片。如前所述，使用散列索引作为片键是可以的。

15.4.2　片键的势

不管片键是跳跃增长还是稳定增长，选择一个值会发生变化的键是非常重要的。与索引一样，分片在势比较高的字段上性能更佳。例如，`"logLevel"` 键只拥有 `"DEBUG"`、`"WARN"` 和 `"ERROR"` 这几个值。如用其作为片键，则 MongoDB 最多只能将数据分为三个块（因为片键只拥有三个不同的值）。如果键拥有的值比较少，而且确实希望将这个键用作片键，则可使用该键与另一个拥有多样值的键创建一个复合片键，比如 `"logLevel"` 和 `"timestamp"`。注意，复合片键的势比较高。

15.5　控制数据分发

有时候，自动数据分发无法满足需求。前面已经学习过了有关选择片键以及让 MongoDB 自动处理事务的内容，接下来我们将在本节学习到更多相关内容。

随着集群变得越来越大或者越来越繁忙，这些解决方案可能会变得不是那么有效。但是，对于比较小的集群，也许我们会希望拥有更多的控制权。

15.5.1　对多个数据库和集合使用一个集群

MongoDB 将集合均衡地分发到集群中的分片上，如果保存的数据比较均匀，则该

方法非常有效。然而，如果有一个日志集合，该集合的数据不如其他集合的数据有“价值”，我们可能不希望其占用昂贵的服务器。或者，如果拥有一个强大的分片，我们可能只希望将其用在实时集合上，而不允许其他集合使用它。这些情况下，可建立独立的集群，也可将数据的保存位置明确指定给 MongoDB。

为实现这种模式，在 shell 中运行 `sh.addShardTag()` 辅助函数：

```
> sh.addShardTag("shard0000", "high")
> // shard0001 - no tag
> // shard0002 - no tag
> // shard0003 - no tag
> sh.addShardTag("shard0004", "low")
> sh.addShardTag("shard0005", "low")
```

然后可以将不同的集合指定到不同的分片。例如，对于实时集合：

```
> sh.addTagRange("super.important", {"shardKey" : MinKey},
... {"shardKey" : MaxKey}, "high")
```

上面这条命令的意思是，“将该集合内片键的值在负无穷到正无穷之间的数据，保存到标签为 `high` 的分片上”。也就是说，该重要集合的所有数据都不会被保存在其他服务器上。注意，这并不会影响其他集合的分发方式：其他集合仍会被均衡地分发在该分片和其他分片上。

同样地，也可以将日志集合指定到比较便宜的服务器上：

```
> sh.addTagRange("some.logs", {"shardKey" : MinKey},
... {"shardKey" : MaxKey}, "low")
```

现在，日志集合会被均衡地分发在 shard0004 和 shard0005 上。

为集合指定一个标签范围的指令并不会立即生效。它只是一个对于均衡器的指令，运行指令可将集合移动到这些目标分片上。因此，如果整个日志集合都位于 shard0002 或者是均衡地分发在所有分片上，那么需要消耗一定的时间，日志集合的所有块才会被迁移到 shard0004 和 shard0005 上。

再举一个例子。也许有这样一个集合，我们希望其出现在除标签为 `high` 的分片以外的任何分片上。可为所有的非高性能分片添加一个新的标签，创建一个新分组。分片可创建的标签多少没有限制：

```
> sh.addShardTag("shard0001", "whatever")
> sh.addShardTag("shard0002", "whatever")
> sh.addShardTag("shard0003", "whatever")
> sh.addShardTag("shard0004", "whatever")
> sh.addShardTag("shard0005", "whatever")
```

现在，可指定该集合（名为 normal.coll）分发在这五个分片上：

```
> sh.addTagRange("normal.coll", {"shardKey" : MinKey},
... {"shardKey" : MaxKey}, "whatever")
```

不能动态地指定集合，如“当新集合创建时，将其随机分发到一个分片上”。但是，可使用定时任务来做这些事情。

如果操作失误或是改变了主意，可使用 sh.removeShardTab() 删除分片的标签：

```
> sh.removeShardTag("shard0005", "whatever")
```

如果删除了某个标签范围的所有标签（例如，删除了标签为 "high" 的分片标签），均衡器不会再将数据分发到任何地方，因为没有可用的位置。所有数据仍可读可写，但不会被迁移到其他位置，除非修改标签或者标签范围。

不存在用于删除标签范围的辅助函数，但可手动删除。手动处理标签范围，可通过 mongos 访问 config.tags 命名空间。类似地，分片的标签信息保存在分片文档 "tags" 字段下的 config.shards 命名空间。如分片文档中没有 "tags" 字段，则该分片就不存在标签。

15.5.2　手动分片

有时候，对于复杂的需求或是特殊的情况，我们可能希望对集群的数据分发拥有绝对控制权。如果不希望数据被自动分发，可关闭均衡器，使用 moveChunk 命令手动对数据进行迁移。

要关闭均衡器，可连接到一个 mongos（任何 mongos 都可以），然后使用以下命令更新 config.settings 命名空间：

```
> db.settings.update({"_id" : "balancer"}, {"enabled" : false}, true)
```

注意，这是一个 upsert 操作：如果均衡器设置不存在，则会自动创建一个。

如正在进行迁移，则该设置要等到当前迁移完成之后才会生效。然而，一旦当前迁移完成了，均衡器就不会再做数据移动了。

只要均衡器被关闭，就可以手动做数据迁移了（如有必要的话）。首先，查看 config.chunks 找出每个块的分发位置：

```
> db.chunks.find()
```

现在，使用 moveChunk 命令将块迁移到其他分片。指定需被迁移块的下边界值和目

标分片的名称：

```
> sh.moveChunk("test.manual.stuff",
... {user_id: NumberLong("-1844674407370955160")}, "test-rs1")
{ "millis" : 4079, "ok" : 1 }
```

然而，除非遇到特殊情况，否则都应使用 MongoDB 的自动分片，而非手动进行分片。如果最后得到一个拥有一个热点的分片（这并非是我们所期望的），那么大部分数据可能都将出现在这个分片上。

尤其不要在均衡器开启的情况下手动做一些不寻常的分发。如果均衡器检测到一些不均衡的块，则会对调整过的数据进行重新分发，以便让集合再次处于均衡状态。如果希望得到非均衡的数据块分发，应使用上一小节介绍过的分片标签技术。

第 16 章
分片管理

对数据库管理员来说，分片集群是最困难的部署类型。本章我们将学习在集群上执行管理任务的方方面面，内容包括：

- 检查集群状态：集群有哪些成员？数据保存在哪里？哪些连接是打开的？
- 如何添加、删除和修改集群的成员；
- 管理数据移动和手动移动数据 。

16.1 检查集群状态

有一些辅助函数可用于找出数据保存位置、所在分片，以及集群正在进行的操作。

16.1.1 使用sh.status查看集群摘要信息

使用 sh.status() 可查看分片、数据库和分片集合的摘要信息。如果块的数量较少，则该命令会打印出每个块的保存位置。否则它只会简单地给出集合的片键，以及每个分片的块数：

```
> sh.status()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
    { "_id" : "shard0000", "host" : "localhost:30000",
      "tags" : [ "USPS" , "Apple" ] }
    {  "_id" : "shard0001", "host" : "localhost:30001" }
    {  "_id" : "shard0002", "host" : "localhost:30002", "tags" : [ "Apple" ] }
```

```
databases:
  {  "_id" : "admin", "partitioned" : false, "primary" : "conÑg" }
  {  "_id" : "test", "partitioned" : true, "primary" : "shard0001" }
    test.foo
      shard key: { "x" : 1, "y" : 1 }
      chunks:
        shard0000    4
        shard0002    4
        shard0001    4
      { "x" : { $minKey : 1 }, "y" : { $minKey : 1 } } -->>
          { "x" : 0, "y" : 10000 } on : shard0000
      { "x" : 0, "y" : 10000 } -->> { "x" : 12208, "y" : -2208 }
          on : shard0002
      { "x" : 12208, "y" : -2208 } -->> { "x" : 24123, "y" : -14123 }
          on : shard0000
      { "x" : 24123, "y" : -14123 } -->> { "x" : 39467, "y" : -29467 }
          on : shard0002
      { "x" : 39467, "y" : -29467 } -->> { "x" : 51382, "y" : -41382 }
          on : shard0000
      { "x" : 51382, "y" : -41382 } -->> { "x" : 64897, "y" : -54897 }
          on : shard0002
      { "x" : 64897, "y" : -54897 } -->> { "x" : 76812, "y" : -66812 }
          on : shard0000
      { "x" : 76812, "y" : -66812 } -->> { "x" : 92793, "y" : -82793 }
          on : shard0002
      { "x" : 92793, "y" : -82793 } -->> { "x" : 119599, "y" : -109599 }
          on : shard0001
      { "x" : 119599, "y" : -109599 } -->> { "x" : 147099, "y" : -137099 }
          on : shard0001
      { "x" : 147099, "y" : -137099 } -->> { "x" : 173932, "y" : -163932 }
          on : shard0001
      { "x" : 173932, "y" : -163932 } -->>
          { "x" : { $maxKey : 1 }, "y" : { $maxKey : 1 } } on : shard0001
    test.ips
      shard key: { "ip" : 1 }
      chunks:
        shard0000    2
        shard0002    3
        shard0001    3
      { "ip" : { $minKey : 1 } } -->> { "ip" : "002.075.101.096" }
        on : shard0000
      { "ip" : "002.075.101.096" } -->> { "ip" : "022.089.076.022" }
        on : shard0002
      { "ip" : "022.089.076.022" } -->> { "ip" : "038.041.058.074" }
        on : shard0002
      { "ip" : "038.041.058.074" } -->> { "ip" : "055.081.104.118" }
        on : shard0002
      { "ip" : "055.081.104.118" } -->> { "ip" : "072.034.009.012" }
        on : shard0000
      { "ip" : "072.034.009.012" } -->> { "ip" : "090.118.120.031" }
        on : shard0001
      { "ip" : "090.118.120.031" } -->> { "ip" : "127.126.116.125" }
        on : shard0001
      { "ip" : "127.126.116.125" } -->> { "ip" : { $maxKey : 1 } }
        on : shard0001
```

```
        tag: Apple { "ip" : "017.000.000.000" } -->> { "ip" : "018.000.000.000" }
        tag: USPS { "ip" : "056.000.000.000" } -->> { "ip" : "057.000.000.000" }
   { "_id" : "test2", "partitioned" : false, "primary" : "shard0002" }
```

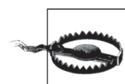
块的数量较多时，`sh.status()` 命令会概述块的状态，而非打印出每个块的相关信息。如需查看所有的块，可使用 `sh.status(true)` 命令（`true` 参数要求 `sh.status()` 命令打印出尽可能详尽的信息）。

`sh.status()` 显示的所有信息都来自 config 数据库。

运行 `sh.status()` 命令，使 MapReduce 获取这一数据，因此，如果启动数据库时指定了 `--noscripting` 选项，则无法运行 `sh.status()` 命令。

16.1.2　检查配置信息

集群相关的所有配置信息都保存在配置服务器上 config 数据库的集合中。可直接访问该数据库，不过 shell 提供了一些辅助函数，并通过这些函数获取更适于阅读的信息。不过，可始终通过直接查询 config 数据库的方式，获取集群的元数据。



永远不要直接连接到配置服务器，以防配置服务器数据被不小心修改或删除。应先连接到 mongos，然后通过 config 数据库来查询相关信息，方法与查询其他数据库一样：

```
mongos> use config
```

如果通过 mongos 操作配置数据（而不是直接连接到配置服务器），mongos 会保证将修改同步到所有配置服务器，也会防止危险操作的发生，如意外删除 config 数据库等。

总的来说，不应直接修改 config 数据库的任何数据（例外情况下面会提到）。如果确实修改了某些数据，通常需要重启所有的 mongos 服务器，才能看到效果。

config 数据库中有一些集合，本节将介绍这些集合的内容和使用方法。

1. config.shards

shards 集合跟踪记录集群内所有分片的信息。shards 集合中的一个典型文档结构如下：

```
> db.shards.findOne()
{
    "_id" : "spock",
    "host" : "spock/server-1:27017,server-2:27017,server-3:27017",
    "tags" : [
        "us-east",
        "64gb mem",
        "cpu3"
```

```
        ]
    }
```

分片的 "_id" 来自于副本集的名称，所以集群中的每个副本集名称都必须是唯一的。

更新副本集配置的时候（比如添加或删除成员），host 字段会自动更新。

2. onfig.databases

databases 集合跟踪记录集群中所有数据库的信息，不管数据库有没有被分片：

```
> db.databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test1", "partitioned" : true, "primary" : "spock" }
{ "_id" : "test2", "partitioned" : false, "primary" : "bones" }
```

如果在数据库上执行过 enableSharding，则此处的 "partitioned" 字段值就是 true。"primary" 是“主数据库”（home base）。数据库的所有新集合均默认被创建在数据库的主分片上。

3. config.collections

collections 集合跟踪记录所有分片集合的信息（非分片集合信息除外）。其中的文档结构如下：

```
> db.collections.findOne()
{
    "_id" : "test.foo",
    "lastmod" : ISODate("1970-01-16T17:53:52.934Z"),
    "dropped" : false,
    "key" : { "x" : 1, "y" : 1 },
    "unique" : true
}
```

下面是一些重要字段。

- _id
  集合的命名空间。

- key
  片键。本例中指由 x 和 y 组成的复合片键。

- unique
  表明片键是一个唯一索引。该字段只有当值为 true 时才会出现（表明片键是唯一的）。片键默认不是唯一的。

4. config.chunks

chunks 集合记录有集合中所有块的信息。chunks 集合中的一个典型文档结构如下所示：

```
{
    "_id" : "test.hashy-user_id_-1034308116544453153",
    "lastmod" : { "t" : 5000, "i" : 50 },
    "lastmodEpoch" : ObjectId("50f5c648866900ccb6ed7c88"),
    "ns" : "test.hashy",
    "min" : { "user_id" : NumberLong("-1034308116544453153") },
    "max" : { "user_id" : NumberLong("-732765964052501510") },
    "shard" : "test-rs2"
}
```

下面这些字段最为有用。

- `_id`
  块的唯一标识符。该标识符通常由命名空间、片键和块的下边界值组成。

- `ns`
  块所属的集合名称。

- `in`
  块范围的最小值（包含）。

- `max`
  块范围的最大值（不包含）。

- `shard`
  块所属的分片。

这里的 `lastmod` 和 `lastmodEpoch` 字段用于记录块的版本。例如，如一个名为 `foo.bar-_id-1` 的块被拆分为两个块，原本的 `foo.bar-_id-1` 会成为一个较小的新块，我们需要一种方式来区别该块与之前的块。因此，我们用 `t` 和 `i` 字段表示块的主（major）版本和副（minor）版本：主版本会在块被迁移至新的分片时发生改变，副版本会在块被拆分时发生改变。

`sh.status()` 获取的大部分信息都来自于 config.chunks 集合。

5. config.changelog

changelog 集合可用于跟踪记录集群的操作，因为该集合会记录所有的拆分和迁移操作。

拆分记录的文档结构如下：

```
{
    "_id" : "router1-2013-02-09T18:08:12-5116908cab10a03b0cd748c3",
    "server" : "spock-01",
    "clientAddr" : "10.3.1.71:62813",
```

```
        "time" : ISODate("2013-02-09T18:08:12.574Z"),
        "what" : "split",
        "ns" : "test.foo",
        "details" : {
            "before" : {
                "min" : { "x" : { $minKey : 1 }, "y" : { $minKey : 1 } },
                "max" : { "x" : { $maxKey : 1 }, "y" : { $maxKey : 1 } },
                "lastmod" : Timestamp(1000, 0),
                "lastmodEpoch" : ObjectId("000000000000000000000000")
            },
            "left" : {
                "min" : { "x" : { $minKey : 1 }, "y" : { $minKey : 1 } },
                "max" : { "x" : 0, "y" : 10000 },
                "lastmod" : Timestamp(1000, 1),
                "lastmodEpoch" : ObjectId("000000000000000000000000")
            },
            "right" : {
                "min" : { "x" : 0, "y" : 10000 },
                "max" : { "x" : { $maxKey : 1 }, "y" : { $maxKey : 1 } },
                "lastmod" : Timestamp(1000, 2),
                "lastmodEpoch" : ObjectId("000000000000000000000000")
            }
        }
    }
```

从 `details` 字段中可以看到文档在拆分前和拆分后的内容。

这里显示的是集合第一个块被拆分后的情景。注意，每个新块的副版本都发生了增长：新块的 `lastmod` 分别是 `Timestamp(1000，1)` 和 `Timestamp(1000，2)`.

数据迁移的操作比较复杂，每次迁移实际上会创建 4 个独立的 changelog 文档：一条是迁移开始时的状态，一条是 `from` 分片的文档，一条是 `to` 分片的文档，还有一条是迁移完成时的状态。中间的两个文档比较有参考价值，因为可从中看出每一步操作耗时多久。这样就可得知，造成迁移瓶颈的到底是磁盘、网络还是其他什么原因了。

例如，`from` 分片的文档结构如下：

```
    {
        "_id" : "router1-2013-02-09T18:15:14-5116923271b903e42184211c",
        "server" : "spock-01",
        "clientAddr" : "10.3.1.71:27017",
        "time" : ISODate("2013-02-09T18:15:14.388Z"),
        "what" : "moveChunk.to",
        "ns" : "test.foo",
        "details" : {
            "min" : { "x" : 24123, "y" : -14123 },
            "max" : { "x" : 39467, "y" : -29467 },
            "step1 of 5" : 0,
            "step2 of 5" : 0,
```

```
            "step3 of 5" : 900,
            "step4 of 5" : 0,
            "step5 of 5" : 142
        }
    };
```

details 字段中的每一步表示的都是时间，stepN of 5 信息以毫秒为单位，显示了步骤的耗时长短。当 from 分片收到 mongos 发来的 moveChunk 命令时，它会：

(1) 检查命令的参数；

(2) 向配置服务器申请获得一个分布锁，以便进入迁移过程；

(3) 尝试连接到 to 分片；

(4) 数据复制，这是整个过程的"临界区"（critical section）；

(5) 与 to 分片和配置服务器一起确认迁移是否成功完成。

注意，step4 of 5 中的 to 和 from 分片间进行的是直接通信：每个分片都是直接连接到另一个分片和配置服务器上，以进行迁移。如果 from 分片在迁移过程的最后一步出现短暂的网络连接问题，它可能会处于无法撤销迁移操作也无法继续进行下去的状态。在这种情况下，mongod 会关闭。

to 分片的 changloe 文档与 from 分片类似，但步骤有些许不同：

```
    {
        "_id" : "router1-2013-02-09T18:15:14-51169232ab10a03b0cd748e5",
        "server" : "spock-01",
        "clientAddr" : "10.3.1.71:62813",
        "time" : ISODate("2013-02-09T18:15:14.391Z"),
        "what" : "moveChunk.from",
        "ns" : "test.foo",
        "details" : {
             "min" : { "x" : 24123, "y" : -14123 },
             "max" : { "x" : 39467, "y" : -29467 },
             "step1 of 6" : 0,
             "step2 of 6" : 2,
             "step3 of 6" : 33,
             "step4 of 6" : 1032,
             "step5 of 6" : 12,
             "step6 of 6" : 0
        }
    }
```

当 to 分片收到 from 分片发来的命令时，它会执行如下操作。

(1) 迁移索引。如果该分片不包含任何来自迁移集合的块，则需知道有哪些字段上建立过索引。如果在此之前 to 分片已有来自于该集合的块，则可忽略此步骤。

(2) 删除块范围内已存在的任何数据。之前失败的迁移（如果有的话）可能会留有数据残余，或者是正处于恢复过程当中，此时我们不希望残留数据与新数据混杂在一起。

(3) 将块中的所有文档复制到 `to` 分片。

(4) 复制期间，在 `to` 分片上重新执行曾在这些文档上执行过的操作。

(5) 等待 `to` 分片将新迁移过来的数据复制到集群的大多数服务器上。

(6) 修改块的元数据以完成迁移过程，表明数据已被成功迁移到 `to` 分片上 。

6. config.tags

该集合的创建是在为系统配置分片标签时发生的。每个标签都与一个块范围相关联：

```
> db.tags.find()
{
    "_id" : {
        "ns" : "test.ips",
        "min" : {"ip" : "056.000.000.000"}
    },
    "ns" : "test.ips",
    "min" : {"ip" : "056.000.000.000"},
    "max" : {"ip" : "057.000.000.000"},
    "tag" : "USPS"
}
{
    "_id" : {
        "ns" : "test.ips",
        "min" : {"ip" : "017.000.000.000"}
    },
    "ns" : "test.ips",
    "min" : {"ip" : "017.000.000.000"},
    "max" : {"ip" : "018.000.000.000"},
    "tag" : "Apple"
}
```

7. onfig.settings

该集合含有当前的均衡器设置和块大小的文档信息。通过修改该集合的文档，可开启或关闭均衡器，也可以修改块的大小。注意，应总是连接到 mongos 修改该集合的值，而不应直接连接到配置服务器进行修改。

16.2 查看网络连接

集群的各组成部分间存在大量的连接。本节我们将学习与分片相关的连接信息。网络信息会在第 23 章详细介绍。

16.2.1 查看连接统计

可使用 `connPoolStats` 命令，查看 mongos 和 mongod 之间的连接信息，并可得知服务器上打开的所有连接：

```
> db.adminCommand({"connPoolStats" : 1})
{
    "createdByType": {
        "sync": 857,
        "set": 4
    },
    "numDBClientConnection": 35,
    "numAScopedConnection": 0,
    "hosts": {
        "config-01:10005,config-02:10005,config-03:10005": {
            "created": 857,
            "available": 2
        },
        "spock/spock-01:10005,spock-02:10005,spock-03:10005": {
            "created": 4,
            "available": 1
        }
    },
    "totalAvailable": 3,
    "totalCreated": 861,
    "ok": 1
}
```

形如 "*host1,host2,host3*" 的主机名是来自配置服务器的连接，也就是用于"同步"的连接。形如 "*name/host1,host2,...,hostN*" 的主机是来自分片的连接。`available` 的值表明当前实例的连接池中有多少可用连接。

注意，只有在分片内的 mongos 和 mongod 上运行这个命令才会有效。

在一个分片上执行 `connPoolStats`，输出信息中可看到该分片与其他分片间的连接，包括连接到其他分片做数据迁移的连接。分片的主连接会直接连接到另一分片的主连接上，然后从目标分片吸取数据。

进行迁移时，分片会建立一个 `ReplicaSetMonitor`（该进程用于监控副本集的健康状况），用于追踪记录迁移另一端分片的健康状况。由于 mongod 不会销毁这个监控器，所以有时会在一个副本集的日志中看到其他副本集成员的信息。这是很正常的，不会对应用程序造成任何影响。

16.2.2 限制连接数量

当有客户端连接到 mongos 时，mongos 会创建一个连接，该连接应至少连接到一个分片上，以便将客户端请求发送给分片。因此，每个连接到 mongos 的客户端连接

都会至少产生一个从 mongos 到分片的连接。

如果有多个 mongos 进程，可能会创建出非常多的连接，甚至超出分片的处理能力：一个 mongos 最多允许 20 000 个连接（mongod 也是如此）。如果有 5 个 mongos 进程，每个 mongos 有 10 000 个客户端连接，那么这些 mongos 可能会试图创建 50 000 个到分片的连接！

为防止这种情况的发生，可在 mongos 的命令行配置中使用 `maxConns` 选项，这样可以限制 mongos 能够创建的连接数量。可使用下列公式计算分片能够处理的来自单一 mongos 的连接数量：

`maxConns`=20 000 －（mongos 进程的数量 ×3）－（每个副本集的成员数量 ×3）－（其他 /mongos 进程的数量）

以下为公式的相关说明。

- (mongos进程的数量×3)
  每个 mongos 会为每个 `mongod` 创建 3 个连接：一个用于转发客户端请求，一个用于追踪错误信息，即写回监听器（writeback listener），一个用于监控副本集状态。

- (每个副本集的成员数量×3)
  主节点会与每个备份节点创建一个连接，而每个备份节点会与主节点创建两个连接，因此总共是 3 个连接。

- (其他 /mongos进程的数量)
  这里的其他指其他可能连接到 mongod 的进程数量，这种连接包括 MMS 代理、shell 的直接连接（管理员用），或者是迁移时连接到其他分片的连接。

注意，`maxConns` 只会阻止 mongos 创建多于 `maxConns` 数量的连接，但并不会帮助处理连接耗尽的问题。连接耗尽时，请求会发生阻塞，等待某些连接被释放。因此，必须防止应用程序使用超过 `maxConns` 数量的连接，尤其是在 mongos 进程数量不断增加时。

MongoDB 实例在安全退出时，会在终止运行之前关闭所有连接。已经连接到 MongoDB 的成员会立即收到套接字错误（socket error），并能够重新刷新连接。但是，如果 MongoDB 实例由于断电、崩溃或者网络问题突然离线，那些已经打开的套接字很可能没有被关闭。在这种情况下，集群内的其他服务器很可能会认为这个 MongoDB 实例仍在有效运转，但是当试图在该 MongoDB 实例上执行操作时，就会遇到错误，继而刷新连接（如果此时该 MongoDB 实例再次上线且运转正常的话）。

连接数量较少时，可快速检测到某台 MongoDB 实例是否已离线。但是，当有成千上万个连接时，每个连接都需要经历被尝试、检测失败，并重新建立连接的过程，此过程中会得到大量的错误。在出现大量重新连接时，除了重启进程，没有其他特殊有效的方法。

16.3 服务器管理

随着集群的增长，我们可能需要增加集群容量或者是修改集群配置。本节我们将学习向集群添加服务器以及从集群删除服务器的方法。

16.3.1 添加服务器

可随时向集群中添加新的 mongos。只要保证在 mongos 的 `--configdb` 选项中指定了一组正确的配置服务，mongos 即可立即与客户端建立连接。

如 14 章所示，可使用 `addShard` 命令，向集群添加新分片。

16.3.2 修改分片的服务器

使用分片集群时，我们可能会希望修改某单独分片的服务器。要修改分片的成员，需直接连接到分片的主服务器上（而不是通过 mongos），然后对副本集进行重新配置。集群配置会自动检测更改，并将其更新到 config.shards 上。不要手动修改 config.shards。

只有在使用单机服务器作为分片，而不是使用副本集作为分片时，才需手动修改 config.shards。

将单机服务器分片修改为副本集分片

最简单的方式是添加一个新的空副本集分片，然后移除单机服务器分片（参见 16.3.3 节）。

如果希望把单机服务器分片转换为副本集分片，过程会复杂得多，而且需要停机。

(1) 停止向系统发送请求。

(2) 关闭单机服务器（这里称其为 server-1）和所有的 mongos 进程。

(3) 以副本集模式重启 server-1（使用 `--replSet` 选项）。

(4) 连接到 server-1，将其作为一个单成员副本集进行初始化。

(5) 连接到配置服务器，替换该分片的入口，在 config.shards 中将分片名称替换为 *`setName`*`/server-1:27017` 的形式。确保三个配置服务器都拥有相同的配置信

息。手动修改配置服务器是有风险的！

可在每个配置服务器上执行 dbhash 命令，以确保配置信息相同：

```
> db.runCommand({"dbhash" : 1})
```

这样可以得到每个集合的 MD5 散列值。不同配置服务器上，config 数据库的集合可能会有所不同，但 config.shards 应始终保持一致。

(6) 重启所有 mongos 进程。它们会在启动时从配置服务器读取分片数据，然后将副本集当作分片对待。

(7) 重启所有分片的主服务器，刷新其配置数据。

(8) 再次向系统发送请求。

(9) 向 server-1 副本集中添加新成员。

这一过程非常复杂，而且很容易出错，因此不建议使用。应尽可能地将空的副本集作为新的分片添加到集群中，数据迁移的事情交给集群去做就好了。

16.3.3 删除分片

通常来说，不应从集群中删除分片。如果经常在集群中添加和删除分片，会给系统带来很多不必要的压力。如果向集群中添加了过多的分片，最好是什么也不做，系统早晚会用到这些分片，而不应该将多余的分片删掉，等以后需要的时候再将其重新添加到集群中。不过，在必要的情况下，是可以删除分片的。

首先保证均衡器是开启的。在排出数据（draining）的过程中，均衡器会负责将待删除分片的数据迁移至其他分片。执行 removeShard 命令，开始排出数据。removeShard 将待删除分片的名称作为参数，然后将该分片上的所有块都移至其他分片上：

```
> db.adminCommand({"removeShard" : "test-rs3"})
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "test-rs3",
    "note" : "you need to drop or movePrimary these databases",
    "dbsToMove" : [
        "blog",
        "music",
        "prod"
    ],
    "ok" : 1
}
```

如果分片上的块较多，或者有较大的块需要移动，排出数据的过程可能会耗时更长。如果存在特大块（jumbo chunk，参见 16.4.4 节），可能需临时提高其他分片的块大小，以便能够将特大块迁移到其他分片。

如需查看哪些块已完成迁移，可再次执行 `removeShard` 命令，查看当前状态：

```
> db.adminCommand({"removeShard" : "test-rs3"})
{
    "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : NumberLong(5),
        "dbs" : NumberLong(0)
    },
    "ok" : 1
}
```

在一个处于排出数据过程的分片上，可执行 `removeShard` 任意多次。

块在移动前可能需要被拆分，所以有可能会看到系统中的块数量在排出数据时发生了增长。假设有一个拥有 5 个分片的集群，块的分布如下：

```
test-rs0    10
test-rs1    10
test-rs2    10
test-rs3    11
test-rs4    11
```

该集群共有 52 个块。如果删除 test-rs3 分片，最终的结果可能会是：

```
test-rs0    15
test-rs1    15
test-rs2    15
test-rs4    15
```

集群现在拥有 60 个块，其中 18 个来自 test-rs3 分片（原本有 11 个，还有 7 个是在排出数据的过程中创建的）。

所有的块都完成迁移后，如果仍有数据库将该分片作为主分片，需在删除分片前将这些数据库移除掉。`removeShard` 命令的输出结果可能如下：

```
> db.adminCommand({"removeShard" : "test-rs3"})
{
    "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : NumberLong(0),
        "dbs" : NumberLong(3)
    },
```

```
    "note" : "you need to drop or movePrimary these databases",
    "dbsToMove" : [
        "blog",
        "music",
        "prod"
    ],
    "ok" : 1
}
```

为完成分片的删除，需先使用 movePrimary 命令将这些数据库移走：

```
> db.adminCommand({"movePrimary" : "blog", "to" : "test-rs4"})
{
    "primary " : "test-rs4:test-rs4/ubuntu:31500,ubuntu:31501,ubuntu:31502",
    "ok" : 1
}
```

然后再次执行 removeShard 命令：

```
> db.adminCommand({"removeShard" : "test-rs3"})
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "test-rs3",
    "ok" : 1
}
```

最后一步不是必需的，但可确保已确实完成了分片的删除。如果不存在将该分片作为主分片的数据库，则块的迁移完成后，即可看到分片删除成功的输出信息。

注意，如果分片开始排出数据，就没有内置办法停止这一过程了。

16.3.4　修改配置服务器

修改配置服务器是非常困难的，而且有风险，通常还需要停机。注意，修改配置服务器前，应做好备份。

在运行期间，所有 mongos 进程的 --configdb 选项值都必须相同。因此，要修改配置服务器，首先必须关闭所有的 mongos 进程（mongos 进程在使用旧的 --configdb 参数时，无法继续保持运行状态），然后使用新的 --configdb 参数重启所有 mongos 进程。

例如，将一台配置服务器增至三台是最常见的任务之一。为实现此操作，首先应关闭所有的 mongos 进程、配置服务器，以及所有的分片。然后将配置服务器的数据目录复制到两台新的配置服务器上（这样三台配置服务器就可以拥有完全相同的数据目录）。接着，启动这三台配置服务器和所有分片。然后，将 --configdb 选项指定为这三台配置服务器，最后重启所有的 mongos 进程。

16.4 数据均衡

通常来说，MongoDB 会自动处理数据均衡。本节我们将学习如何启用和禁用自动均衡，以及如何人为干涉均衡过程。

16.4.1 均衡器

在执行几乎所有的数据库管理操作之前，都应先关闭均衡器。可使用下列 shell 辅助函数关闭均衡器：

```
> sh.setBalancerState(false)
```

均衡器关闭后，系统则不会再进入均衡过程，但该命令并不能立即终止进行中的均衡过程：迁移过程通常无法立即停止。因此，应检查 config.locks 集合，以查看均衡过程是否仍在进行中：

```
> db.locks.find({"_id" : "balancer"})["state"]
0
```

此处的 0 表明均衡器已被关闭。可翻阅 14.4 节查看均衡器状态相关内容。

均衡过程会增加系统负载：目标分片必须查询源分片块中的所有文档，将文档插入目标分片的块中，源分片最后必须删除这些文档。在以下两种特殊情况下，迁移会导致性能问题。

(1) 使用热点片键可保证定期迁移（因为所有的新块都是创建在热点上的）。系统必须有能力处理源源不断写入到热点分片上的数据。
(2) 向集群中添加新的分片时，均衡器会试图为该分片写入数据，从而触发一系列的迁移过程。

如果发现数据迁移过程影响了应用程序性能，可在 config.settings 集合中为数据均衡指定一个时间窗口。执行下列更新语句，均衡则只会在下午 1 点到 4 点间发生：

```
> db.settings.update({"_id" : "balancer"},
... {"$set" : {"activeWindow" : {"start" : "13:00", "stop" : "16:00"}}},
... true )
```

如指定了均衡时间窗，则应对其进行严密监控，以确保 mongos 确实只在指定的时间内做均衡。

如需混用手动均衡和自动均衡，必须格外小心。因为自动均衡器总是根据数据集的当前状态来决定数据迁移，而不考虑数据集的历史状态。例如，假设有两个分片 shardA 和 shardB，每个分片都有 500 个块。由于 shardA 上的写请求比较多，因此

我们关闭了均衡器，从最活跃的块中取出 30 个移至 shardB。此时如再启用均衡器，则会立即将 30 个块（很可能不是刚刚的 30 块）从 shardB 移至 shardA，以均衡两个分片拥有的块数量。

为防止这种情况发生，可在启用均衡器之前从 shardB 选取 30 个不活跃的块移至 shardA。这样两个分片间就不会存在不均衡，均衡器也不会进行数据块的移动了。另外，也可在 shardA 上拆分出一些块，以实现 shardA 和 shardB 的均衡。

注意，均衡器只使用块的数量，而非数据大小，作为衡量分片间是否均衡的指标。因此，如果 A 分片只拥有几个较大的数据块，而 B 分片拥有许多较小的块（但总数据大小比 A 小），那么均衡器会将 B 分片的一些块移至 A 分片，从而实现均衡。

16.4.2 修改块大小

块中的文档数量可能为 0，也可能多达数百万。通常情况下，块越大，迁移至分片的耗时就越长。在第 13 章中，我们使用的是 1 MB 的块，所以块移动起来非常容易与迅速。但在实际系统中，这通常是不现实的。MongoDB 需要做大量非必要的工作，才能将分片大小维持在几 MB 以内。块的大小默认为 64 MB，这个大小的块既易于迁移，又不会导致过多的流失。

有时可能会发现移动 64 MB 的块耗时过长。可通过减小块的大小，提高迁移速度。使用 shell 连接到 mongos，然后修改 config.settings 集合，从而完成块大小的修改：

```
> db.settings.findOne()
{
    "_id" : "chunksize",
    "value" : 64
}
> db.settings.save({"_id" : "chunksize", "value" : 32})
```

以上修改操作将块的大小减至 32 MB。已经存在的块不会立即发生改变，执行块拆分操作时，这些块即可拆分成 32 MB 大小。mongos 进程会自动加载新的块大小。

注意，该设置的有效范围是整个集群：它会影响所有集合和数据库。因此，如需对一个集合使用较小的块，而对另一集合使用较大的块，比较好的解决方式是取一个折中的值（或者将这两个集合放在不同的集群中）。

如果 MongoDB 频繁进行数据迁移或文档较大，则可能需要增加块的大小。

16.4.3 移动块

如前所述，同一块内的所有数据都位于同一分片上。如该分片的块数量比其他分片

多，则 MongoDB 会将其中的一部分块移至其他块数量较少的分片上。移动块的过程叫做*迁移*（migration），MongoDB 就是这样在集群中实现数据均衡的。

可在 shell 中使用 `moveChunk` 辅助函数，手动移动块：

```
> sh.moveChunk("test.users", {"user_id" : NumberLong("1844674407370955160")},
... "spock")
{ "millis" : 4079, "ok" : 1 }
```

以上命令会将包含文档 `user_id` 为 1844674407370955160 的块移至名为 `spock` 的分片上。必须使用片键来找出所需移动的块（本例中的片键是 `user_id`）。通常，指定一个块最简单的方式是指定它的下边界，不过指定块范围内的任何值都可以（块的上边界值除外，因为其并不包含在块范围内）。该命令在块移动完成后才会返回，因此需一定耗时才能看到输出信息。如某个操作耗时较长，可在日志中详细查看问题所在。

如某个块的大小超出了系统指定的最大值，mongos 则会拒绝移动这个块：

```
> sh.moveChunk("test.users", {"user_id" : NumberLong("1844674407370955160")},
... "spock")
{
    "cause" : {
        "chunkTooBig" : true,
        "estimatedChunkSize" : 2214960,
        "ok" : 0,
        "errmsg" : "chunk too big to move"
    },
    "ok" : 0,
    "errmsg" : "move failed"
}
```

本例中，移动这个块之前，必须先手动拆分这个块。可使用 `splitAt` 命令对块进行拆分：

```
> db.chunks.Ṅnd({"ns" : "test.users",
... "min.user_id" : NumberLong("1844674407370955160")})
{
    "_id" : "test.users-user_id_NumberLong(\"1844674407370955160\")",
    "ns" : "test.users",
    "min" : { "user_id" : NumberLong("1844674407370955160") },
    "max" : { "user_id" : NumberLong("2103288923412120952") },
    "shard" : "test-rs2"
}
> sh.splitAt("test.ips", {"user_id" : NumberLong("2000000000000000000")})
{ "ok" : 1 }
> db.chunks.Ṅnd({"ns" : "test.users",
... "min.user_id" : {"$gt" : NumberLong("1844674407370955160")},
... "max.user_id" : {"$lt" : NumberLong("2103288923412120952")}})
```

```
{
    "_id" : "test.users-user_id_NumberLong(\"1844674407370955160\")",
    "ns" : "test.users",
    "min" : { "user_id" : NumberLong("1844674407370955160") },
    "max" : { "user_id" : NumberLong("2000000000000000000") },
    "shard" : "test-rs2"
}
{
    "_id" : "test.users-user_id_NumberLong(\"2000000000000000000\")",
    "ns" : "test.users",
    "min" : { "user_id" : NumberLong("2000000000000000000") },
    "max" : { "user_id" : NumberLong("2103288923412120952") },
    "shard" : "test-rs2"
}
```

块被拆分为较小的块后，就可以被移动了。也可以调高最大块的大小，然后再移动这个较大的块。不过应尽可能地将大块拆分为小块。不过有时有些块无法被拆分，这些块被称作特大块。

16.4.4 特大块

假设使用 `date` 字段作为片键。集合中的 `date` 字段是一个日期字符串，格式为 *`year/month/day`*，也就是说，mongos 一天最多只能创建一个块。最初的一段时间内一切正常，直到有一天，应用程序的业务量突然出现病毒式增长，流量比平常大了上千倍！

这一天的块要比其他日期的大得多，但由于块内所有文档的片键值都是一样的，因此这个块是不可拆分的。

如果块的大小超出了 config.settings 中设置的最大块大小，那么均衡器就无法移动这个块了。这种不可拆分和移动的块就叫做特大块，这种块相当难对付。

举例来说，假如有 3 个分片 shard1、shard2 和 shard3。如果使用热点片键模式（参见 15.2.1 节），假设 shard1 是热点片键，则所有写请求都会被分发到 shard1 上。mongos 会试图将块均衡地分发在这些分片上。但是，均衡器只能移动非特大块，因此它只会将所有较小块从热点分片迁移到其他分片。

现在，所有分片上的块数基本相同，但 shard2 和 shard3 上的所有块都小于 64 MB。如 shard1 上出现了特大块，则 shard1 上会有越来越多的块大于 64 MB。这样，即使三个分片的块数非常均衡，但 shard1 会比另两个分片更早被填满。

出现特大块的表现之一是，某分片的大小增长速度要比其他分片快得多。也可使用 `sh.status()` 来检查是否出现了特大块：特大块会存在一个 `jumbo` 属性。

```
> sh.status()
...
    { "x" : -7 } -->> { "x" : 5 } on : shard0001
    { "x" : 5 } -->> { "x" : 6 } on : shard0001 jumbo
    { "x" : 6 } -->> { "x" : 7 } on : shard0001 jumbo
    { "x" : 7 } -->> { "x" : 339 } on : shard0001
...
```

可使用 `dataSize` 命令检查块大小。

首先，使用 config.chunks 集合，查看块范围：

```
> use config
> var chunks = db.chunks.find({"ns" : "acme.analytics"}).toArray()
```

然后根据这些块范围，找出可能的特大块：

```
> use dbName
> db.runCommand({"dataSize" : "dbName.collName",
... "keyPattern" : {"date" : 1}, // 片键
... "min" : chunks[0].min,
... "max" : chunks[0].max})
{ "size" : 11270888, "numObjects" : 128081, "millis" : 100, "ok" : 1 }
```

但要小心，因为 `dataSize` 命令要扫描整个块的数据才能知道块的大小。因此如果可能，应首先根据自己对数据的了解，尽可能缩小搜索范围：特大块是在特定日期出现的吗？例如，如果 11 月 1 号的时候系统非常繁忙，则可尝试检查这一天创建的块的片键范围。如使用了 GridFS，而且是依据 `files_id` 字段进行分片的，则可通过 fs.files 集合查看文件大小。

1. 分发特大块

为修复由特大块引发的集群不均衡，就必须将特大块均衡地分发到其他分片上。

这是一个非常复杂的手动过程，而且不应引起停机（可能会导致系统变慢，因为要迁移大量的数据）。接下来，我们以 from 分片来指代拥有特大块的分片，以 to 分片来指代特大块即将移至的目标分片。注意，如有多个 from 分片，则需对每个 from 分片重复下列步骤：

(1) 关闭均衡器，以防其在这一过程中出来捣乱：

```
> sh.setBalancerState(false)
```

(2) MongoDB 不允许移动大小超出最大块大小设定值的块，所以需临时调高最大块大小的设定值。记下特大块的大小，然后将最大块大小设定值调整为比特大块大一些的数值，比如 10 000。块大小的单位是 MB：

```
> use config
> db.settings.findOne({"_id" : "chunksize"})
{
    "_id" : "chunksize",
    "value" : 64
}
> db.settings.save({"_id" : "chunksize", "value" : 10000})
```

(3) 使用 `moveChunk` 命令将特大块从 from 分片移至 to 分片。如担心迁移会对应用程序的性能造成影响，可使用 `secondaryThrootle` 选项，放慢迁移的过程，减缓对系统性能的影响：

```
> db.adminCommand({"moveChunk" : "acme.analytics",
... "find" : {"date" : new Date("10/23/2012")},
... "to" : "shard0002",
... "secondaryThrottle" : true})
```

`secondaryThrottle` 会强制要求迁移过程间歇进行，每迁移完一些数据，需等待集群中的大多数分片成功完成数据复制后再进行下一次迁移。该选项只有在使用副本集分片时才会生效。如使用单机服务器分片，则该选项不会生效。

(4) 使用 `splitChunk` 命令对 from 分片剩余的块进行拆分，这样可以增加 from 分片的块数，直到实现 from 分片与其他分片块数的均衡。

(5) 将块大小修改回最初值：

```
> db.settings.save({"_id" : "chunksize", "value" : 64})
```

(6) 启用均衡器。

```
> sh.setBalancerState(true)
```

均衡器被再次启用后，仍旧不能移动特大块，不过此时那些特大块都已位于合适的位置了。

2. 防止出现特大块

随着存储数据量的增长，上一节提到的手动过程变得不再可行。因此，如在特大块方面存在问题，应首先想办法避免特大块的出现。

为防止特大块的出现，可修改片键，细化片键的粒度。应尽可能保证每个文档都拥有唯一的片键值，或至少不要出现某个片键值的数据块超出最大块大小设定值的情况。

例如，如使用前面所述的年 / 月 / 日片键，可通过添加时、分、秒来细化片键粒度。类似地，如使用粒度较大的片键，如日志级别，则可添加一个粒度较细的字段作为

片键的第二个字段，如 MD5 散列值或 UDID。这样一来，即使有许多文档片键的第一个字段值是相同的，也可一直对块进行拆分，也就防止了特大块的出现。

16.4.5 刷新配置

最后一点，mongos 有时无法从配置服务器正确更新配置。如发现配置有误，mongos 的配置过旧或无法找到应有数据，可使用 `flushRouterConfig` 命令手动刷新所有缓存：

```
> db.adminCommand({"flushRouterConfig" : 1})
```

如 `flushRouterConfig` 命令没能解决问题，则应重启所有的 mongos 或 mongod 进程，以便清除所有可能的缓存。

第五部分

应用管理

第 17 章

了解应用的动态

启动并运行应用后，要如何知道它正在做些什么呢？本章将介绍如何了解 MongoDB 正在进行何种查询，有多少数据正在写入，以及如何探查 MongoDB 具体正在做些什么。我们将学到：

- 如何找到并终止那些拖慢速度的操作；
- 获取并分析有关集合和数据库的统计数据；
- 用命令行工具来了解 MongoDB 正在做些什么。

17.1 了解正在进行的操作

要想找到是哪些操作拖慢了速度，看看正在进行的操作不失为一种简单的方法。速度慢的操作耗时更长，更有可能被发现。虽然不能保证一定会有结果，但这是个不错的开始。

查看正在进行的操作，可使用 `db.currentOp()` 函数：

```
> db.currentOp()
{
    "inprog" : [
        {
            "opid" : 34820,
            "active" : true,
            "secs_running" : 0,
            "op" : "query",
            "ns" : "test.users",
```

```
            "query" : {
                "count" : "users",
                "query" : {
                    "username" : "user12345"
                 },
                 "fields" : {

                 }
            },
            "client" : "127.0.0.1:39931",
            "desc" : "conn3",
            "threadId" : "0x7f12d61c7700",
            "connectionId" : 3,
            "locks" : {
                "^" : "r",
                "^test" : "R"
            },
            "waitingForLock" : false,
            "numYields" : 0,
            "lockStats" : {
                "timeLockedMicros" : {

                },
                "timeAcquiringMicros" : {
                    "r" : NumberLong(9),
                    "w" : NumberLong(0)
                }
            }
        },
        ...
    ]
}
```

该函数会列出数据库正在进行的所有操作，输出的信息中有些重要的字段。

- `opid`

  这是操作的唯一标识符（identifier），可通过它来终止一个操作（参见 17.1.2 节）。

- `active`

  表示该操作是否正在运行。如这一字段的值是 `false`，意味着此操作已交出或正等待其他操作交出锁。

- `secs_running`

  表示该操作已经执行的时间。可通过它来判断是哪些查询耗时过长，或者占用了过多的数据库资源。

- `op`

  表示操作的类型。通常是查询、插入、更新、删除中的一种。注意，数据库命令也被作为查询操作来处理。

- desc
  该值可与日志（log）信息联系起来。日志中与此连接相关的每一条记录都会以 [conn3] 为前缀，因此可以此来筛选相关的日志信息。

- locks
  描述该操作使用的锁的类型。其中“^”表示全局锁。

- waitingForLock
  表示该操作是否因正在等待其他操作交出锁而处于阻塞状态。

- numYields
  表示该操作交出锁（yield），而使其他操作得以运行的次数。通常，进行文档搜索的操作（查询、更新和删除）可交出锁。只有在其他操作列队等待该操作所持的锁时，它才会交出自己的锁。简单地讲，如果没有其他操作处于 waitingForLock 状态，则该操作不会交出锁。

- lockstats.timeAcquiringMicros
  表示该操作需要多长时间才能取得所需的锁。

在执行 currentOp() 时，可添加过滤条件，从而只显示符合条件的结果。例如，只显示在某一命名空间中进行的操作，或只显示已运行了一定时间的操作。把查询条件作为参数传入函数来进行过滤：

```
> db.currentOp({"ns" : "prod.users"})
```

对于 currentOp 中的任何字段都可以进行查询，使用普通的查询语句即可。

17.1.1　寻找有问题的操作

db.currentOp() 最常见的作用就是用来寻找速度较慢的操作。可采用上一节中提到的过滤方法，来查找哪些查询消耗的时间超过了一定的值。也许能通过该方法找出哪里缺少了索引，或是进行了不恰当的条件过滤。

有时会发现正在运行一些不明查询，这通常是由于一个应用服务器在运行一个旧的或有漏洞的软件版本所导致的。"client" 字段可用来帮助追踪找出这些不明操作的来源。

17.1.2　终止操作的执行

只要找到了想要终止的操作，就可将该操作的 opid 作为参数，通过执行 db.killOp() 来终止该操作的执行：

```
> db.killOp(123)
```

并非所有操作都能被终止。一般来讲，只有交出了锁的进程才能被终止，因此更新（update）、查找（find）、删除（remove）操作都可被终止。正在占用锁，或正在等待其他操作交出锁的操作则通常无法被终止。

如果向一个操作发出了“kill”信号，那么它在 `db.currentOp` 的输出中就会有一个 `killed` 字段。然而，只有从当前操作列表消失后，它才会真正的得到终止。

17.1.3　假象

在查找哪些操作耗时过长时，可能会发现一些长时间运行的内部操作。根据设置，MongoDB 可能会长时间地执行若干请求。最常见的是用于复制（replication）的线程（它会持续向同步源请求更多的操作）和分片中用于回写（writeback）的监听器（listener）。所有 `local.oplog.rs` 中的长时间运行请求，以及所有回写监听命令，都可以被忽略掉。

如以上操作被终止，MongoDB 则会重启它们。不过，通常我们不应该这么做。终止用于复制的线程会短暂地中止复制操作，而终止掉回写监听器则可能会造成 mongos 遗漏正常的写入错误。

17.1.4　避免幽灵操作

这是一个不常见的，只有在 MongoDB 中才可能会遇到的问题，尤其是在进行静态加载（bulk-loading）数据至集合的时候。假设现在我们建立了一个任务（job），用于在 MongoDB 中进行上千条更新操作，而 MongoDB 正逐渐趋于停止。我们迅速停止了这一任务，终止了正在进行的所有更新操作。然而，我们会发现新的更新操作不断出现，哪怕任务已经不再运行！

如果使用非应答式写入（unacknowledge write）加载数据，应用触发写入操作的速度可能要比 MongoDB 处理的速度更快。如 MongoDB 有所准备，这些写入会堆积在操作系统的套接字缓存（socket buffer）中。终止掉 MongoDB 正在进行的写入操作后，MongoDB 则开始处理缓存区中的写入操作。即使停止客户端发送，MongoDB 也会处理这些缓存中的写入请求，因为它们已经被 MongoDB 所接收了，只不过还没有进行处理而已。

阻止这些幽灵写入的最好方式是使用应答式写入，即每次写入操作都会等待上一次写入操作完成后才会进行下去，而非在上一次写入进入数据库服务器的缓存区就开始下一次写入。

17.2 使用系统分析器

可利用系统分析器（system profiler）来查找耗时过长的操作。系统分析器可记录特殊集合 system.profile 中的操作，并提供大量有关耗时过长的操作信息，但相应的，mongod 的整体性能也会有所下降。因此，我们可能只需定期打开分析器来获取信息即可。如系统已经负载过重，则建议使用本章介绍的另一方法来解决问题。

默认情况下，系统分析器处于关闭状态，不会进行任何记录。可在 shell 中运行 `db.setProfilingLevel()` 开启分析器：

```
> db.setProfilingLevel(2)
{ "was" : 0, "slowms" : 100, "ok" : 1 }
```

以上命令将分析器的级别设定为 2 级，意味着“分析器会记录所有内容”。数据库收到的所有读写请求都将被记录在当前数据库的 system.profile 集合中。每一个数据库都启用了分析器，这也将带来大量的性能损失，因为每一次写操作都会增加额外的写入时间，而每一次读操作都要等待写锁（因为它必须在 system.profile 集合中写入记录）。然而，它也会提供给我们系统进行操作的详尽列表：

```
> db.foo.insert({x:1})
> db.foo.update({},{$set:{x:2}})
> db.foo.remove()
> db.system.profile.find().pretty()
{
    "ts" : ISODate("2012-11-07T18:32:35.219Z"),
    "op" : "insert",
    "ns" : "test.foo",
    "millis" : 37,
    "client" : "127.0.0.1",
    "user" : ""
}
{
    "ts" : ISODate("2012-11-07T18:32:47.334Z"),
    "op" : "update",
    "ns" : "test.foo",
    "query" : {

    },
    "updateobj" : {
        "$set" : {
            "x" : 2
        }
    },
    "nscanned" : 1,
    "fastmod" : true,
    "millis" : 3,
    "client" : "127.0.0.1",
    "user" : ""
```

```
}
{
    "ts" : ISODate("2012-11-07T18:32:50.058Z"),
    "op" : "remove",
    "ns" : "test.foo",
    "query" : {
    },
    "millis" : 0,
    "client" : "127.0.0.1",
    "user" : ""
}
```

在 "client"（客户端）字段中可看到各操作是由哪个用户发送至数据库的。如果启用了身份验证系统，也能够看到各操作是由哪些用户运行的。

一般情况下，我们只想关注那些耗时过长的操作，而非数据库中正在进行的所有操作。为此，可将分析器的分析级别设为 1，即只显示长耗时操作。级别为 1 的分析器会默认记录耗时大于 100 ms 的操作。也可以自定义"耗时过长"的标准，把这个值作为 db.setProfillingLevel() 函数的第二个参数。以下命令会记录所有耗时超过 500 ms 的操作：

```
> db.setProfilingLevel(1, 500)
{ "was" : 2, "slowms" : 100, "ok" : 1 }
```

将分析级别设为 0 可关闭分析器。

```
> db.setProfilingLevel(0)
{ "was" : 1, "slowms" : 500, "ok" : 1 }
```

通常情况下，不要将 slowms 的值设得过小。即使分析器处于关闭状态，slowms 也会对 mongod 有所影响，因为它决定了哪些操作将作为耗时过长操作被记录到日志中。因此，如果将 slowms 设为 2 ms，那么哪怕分析器是关闭着的，每个耗时超过 2 ms 的操作也都会出现在日志里。因此，如果出于某些需求降低了 slowms 的值，那么应在关闭分析器前将它重新调高。

可通过 db.getProfilingLevel() 来查看当前的分析级别。分析级别的设定值会在重启数据库后被清除。

也可在命令行中使用 --profile level 和 --slowms time 选项来配置分析器的级别。但更改分析级别通常只是在调试时作为一种临时措施，而不应该将其长期地加入配置中。

如开启了分析器而 system.profile 集合并不存在，MongoDB 会为其建立一个大小为若干 MB 的固定集合（capped collection）。如希望分析器运行更长时间，可能需要

更大的空间来记录更多的操作。此时可关闭分析器，删除并重新建立一个新的名为 system.profile 的固定集合，并令其容量符合需求。然后在数据库上重新启用分析器。

17.3 计算空间消耗

如能得知文档、索引、集合、数据库各占用了多少空间，就可以方便地预留出合适的磁盘和内存空间。关于计算工作集大小的相关内容请参见第 21 章。

17.3.1 文档

要查询文档占用的空间大小，最简单的方法是在 shell 中对文档使用 `Object.bsonsize()` 函数。此函数将返回该文档存储在 MongoDB 中时占用的空间大小。

例如，我们可以看到，将 `_id` 存储为 `ObjectId` 类型，比存储为字符串类型效率更高。

```
> Object.bsonsize({_id:ObjectId()})
22
> // ""+ObjectId() 将 ObjectId 转换为字符串
> Object.bsonsize({_id:""+ObjectId()})
39
```

也可以直接对集合中的文档进行查询：

```
> Object.bsonsize(db.users.findOne())
```

这一函数会精确地告知文档在磁盘上占用的字节数目。然而这其中并未包括自动生成的空间间隔（padding）和索引，二者也时常是影响集合大小的重要因素。

17.3.2 集合

`stats` 函数可用来显示一个集合的信息：

```
> db.boards.stats()
{
    "ns" : "brains.boards",
    "count" : 12,
    "size" : 32292,
    "avgObjSize" : 2691,
    "storageSize" : 270336,
    "numExtents" : 3,
    "nindexes" : 2,
    "lastExtentSize" : 212992,
    "paddingFactor" : 1.0099999999999825,
    "flags" : 1,
    "totalIndexSize" : 16352,
    "indexSizes" : {
```

```
        "_id_" : 8176,
        "username_1_slug_1" : 8176
    },
    "ok" : 1
}
```

stats 函数的返回结果中首先是命名空间（即 brains.boards），接下来是集合中文档的数目。再接下来的几个字段与集合的大小有关。size 的值相当于对此集合中的所有元素执行 Object.bsonsize()，再将这些结果相加得到的值，即集合中所有文档占有的字节数。将 avgObjSize（平均对象大小）和 count 相乘，也能得到 size 的值。

与之前提到的一样，所有文档占用的字节总数并不等于集合大小，集合还占用空间存放其他重要内容，即文档间的间隔和索引信息。而 storageSize 不仅包含这些内容，还包含集合两端预留的未经使用空间。集合末端总有些空余空间，以便新文档能够快速添加进来。

nindexes 是集合中索引的数量。索引直到建立完成后才会被算在 nindexes 中，也只有在出现在此列表后才可以被使用。由于目前的集合还很小，所以每个索引都只有一个“桶”（bucket）大小（8 KB）。通常来讲，索引比存储的数据量大很多，含有很多空闲空间，以便在增加新入口（entry）时进行优化。使用右平衡索引（right-balanced index，参见 5.1.1 节）可将这一空闲空间减至最小。而随机分布的索引通常会有 50% 左右的空闲空间，升序索引（ascending-order index）则有 10% 的空闲空间。

随着集合的不断增长，stats() 返回的巨大字节数目可能会变得不易辨识。因此，可在使用 stats 时传入比例因子（scale factor）：KB 值为 1024，MB 则为 1024×1024，依次类推。例如，以下命令会以 TB 为单位显示集合信息：

```
> db.big.stats(1024*1024*1024*1024)
```

17.3.3 数据库

数据库的 stats 函数与集合的类似：

```
> db.stats()
{
    "db" : "brains",
    "collections" : 11,
    "objects" : 1109,
    "avgObjSize" : 299.79440937781783,
    "dataSize" : 332472,
    "storageSize" : 1654784,
```

```
    "numExtents" : 15,
    "indexes" : 11,
    "indexSize" : 114464,
    "fileSize" : 201326592,
    "nsSizeMB" : 16,
    "ok" : 1
}
```

首先返回的是数据库名称和其中包含的集合数目。`objects` 的值是数据库中所有集合包含的文档总数。

输出中包含了有关数据大小的信息。`fileSize` 应该总是最大的，即为数据库文件分配的总空间。该值应等于数据目录中所有名为 brains.* 的文件大小总和。

第二大的字段通常是 `storageSize`，即数据库正在使用的总空间大小。该值与 `fileSize` 不符，因为 `fileSize` 包含了预分配（preallocated）文件。例如，如果数据目录中已经存在 brains.0、brains.1 和 brains.2 文件，则 brains.2 会被 0 填满。brains.2 写入数据后，文件 brains.3 会被预分配。每个数据库内应一直存在一个填充为 0 的空文件。该空文件被写入数据后，下一个文件则会被预分配。因此，该空文件（以及前面文件中未被使用的部分）造成了 `ﬁleSize` 和 `storageSize` 间的差异。

`dataSize` 是此数据库中的数据所占用的空间大小。注意，该值并不包含空闲列表（free list）中的空间，但包含了文档间的间隔。因此该值与 `storageSize` 值的差异，应为被删除文档的大小。。

与集合的 `stats()` 一样，`db.stats()` 可接收一个比例因子作为参数。

如果对一个不存在的数据库使用 `db.stats()`，则 `nsSizeMB` 的值为 0。这是 .ns 文件的大小，它本质上相当于数据库中的内容表。任何存在的数据库均需一个 .ns 文件。

记住，在一个繁忙的系统上列出数据库信息会非常慢，而且会阻碍其他操作。因此应尽量避免此类操作。

17.4 使用mongotop和monogostat

MongoDB 自带了几个命令行工具，可通过每隔几秒输出当前状态，帮助我们判断数据库正在做些什么。

mongotop 类似于 UNIX 中的 top 工具，可概述哪个集合最为繁忙。可通过运行 `mongotop-locks`，从而得知每个数据库的锁状态。

mongostat 提供有关服务器的信息。mongostat 默认每秒输出一次包含当前状态的列

表，可在命令行中传入参数更改时间间隔。每个字段都会给出自上一次被输出以来，所对应的活动发生次数。

- `insert` / `query` / `update` / `delete` / `getmore` / `command`
  每种对应操作的发生次数。
- `flushes`
  mongod 将数据刷新（flush）到磁盘的次数。
- `mapped`
  mongod 所映射的内存数量，通常约等于数据目录的大小。
- `vsize`
  mongod 正在使用的虚拟内存大小，通常为数据目录的 2 倍大小（一次用于映射的文件，一次用于日记系统）。
- `res`
  mongod 正在使用的内存大小，通常该值应尽量接近机器的所有内存大小。
- `locked db`
  在上一个时间片中，锁定时间最长的数据库。该百分比是根据数据库被锁定的时间和全局锁的锁定时间来计算的，这意味着该值可能超过 100%。
- `idx miss %`
  输出中最令人困惑的字段名。指有多少索引在访问中发生了缺页中断（page fault），即索引入口（或被搜索的索引内容）不在内存中，使得 mongod 必须到磁盘中进行读取。
- `qr|qw`
  读写操作的队列（queue）大小，即有多少读写操作被阻塞，等待进行处理。
- `ar|aw`
  指活动客户端的数量，即正在进行读写操作的客户端。
- `netIn`
  通过网络传输进来的字节数，由 MongoDB 进行统计（不必和操作系统的统计相等）。
- `netOut`
  通过网络传输出的字节数，由 MongoDB 进行统计。

- `conn`

  此服务器打开的连接数，包括输入和输出连接。

- `time`

  指以上统计信息所用时间。

可在副本集或分片集群上运行 mongostat。如使用 `--discover` 选项，mongostat 会尝试在初始连接的成员中寻找副本集或分片集群中的所有成员，每台服务器也会每秒针对每个成员输出一行信息。对于较大集群而言，该选项会使数据输出过多过快而不易于管理，但于较小集群而言却很实用，也可使用一些工具将其输出的信息转换为更可读的形式。

要想获得数据库中正在进行的操作快照，mongostat 是很好的选择，但如果要对数据库进行长期的监控，类似 MMS 的工具可能更为适合（参见第 21 章）。

第 18 章

数据管理

本章将学习如何管理集合与数据库。通常来讲，这部分内容并非每天都能用到，但对于应用性能却无比重要，具体包括了：

- 配置用户账户和身份验证；
- 在正在运行的系统中建立索引；
- 对新服务器进行“预热”，以便快速上线；
- 整理数据文件中的碎片；
- 手动预分配新的数据文件。

18.1 配置身份验证

作为系统管理员，首要任务之一就是确保系统安全。确保 MongoDB 安全的最好办法，即在一个可信环境中运行，确保只有可信的机器能够连接到服务器。也就是说，即使是在以任务为颗粒的粗粒度（coarse-grained）访问方式中，MongoDB 也支持针对单个连接进行身份验证。



可登陆 MongoDB 企业版（http://bit.ly/15nFgI3）查看更多复杂的安全特性。在 http://docs.mongodb.org/manual/security 中可找到最新的认证和授权信息。

18.1.1 身份验证基本原理

MongoDB 中，每个数据库的实例都可拥有任意多个用户。安全检查开启后，只有通过身份验证的用户才能够进行数据的读写操作。

admin（管理员）和 local（本地）是两个特殊的数据库，它们当中的用户可对任何数据库进行操作。这两个数据库中的用户可被看作是超级用户。经认证后，管理员用户可对任何数据库进行读写，同时能 执行某些只有管理员才能执行的命令，如 `listDatabases` 和 `shutdown`。

已开启安全检查的数据库在被启动前，应至少添加一个管理员用户。我们来看一个小例子，假设在没有开启安全检查的前提下，已在 shell 中连接到了服务器：

```
> use admin
switched to db admin
> db.addUser("root", "abcd");
{
    "user" : "root",
    "readOnly" : false,
    "pwd" : "1a0f1c3c3aa1d592f490a2addc559383"
}
> use test
switched to db test
> db.addUser("test_user", "efgh");
{
    "user" : "test_user",
    "readOnly" : false,
    "pwd" : "6076b96fc3fe6002c810268702646eec"
}
> db.addUser("read_user", "ijkl", true);
{
    "user" : "read_user",
    "readOnly" : true,
    "pwd" : "f497e180c9dc0655292fee5893c162f1"
}
```

在以上操作中，我们增加了一名管理员用户 `root`，又在名为 `test` 的数据库中增加了两个用户。其中名为 `read_user` 的用户只有读权限而没有写权限。在 shell 中用 `addUser` 创建用户时，将第三个参数 readOnly 设置为 `true`，即可创建一个只读权限用户。运行 `addUser` 时，必须拥有相应数据库的写入权限。这个例子中由于我们还没有启用安全检查，因此可在任一数据库上运行 `addUser`。



除添加新用户外，`addUser` 命令还可用来更改用户密码或只读权限状态。只需在运行 `addUser` 时，将用户名和新密码或只读权限设置作为参数即可。

现在重启服务器，这次在命令行选项中加上 `--auth` 参数，以启用安全检查。启用后，在 shell 中重新连接并尝试以下操作：

```
> use test
switched to db test
> db.test.find();
error: { "$err" : "unauthorized for db [test] lock type: -1 " }
> db.auth("read_user", "ijkl");
1
> db.test.find();
{ "_id" : ObjectId("4bb007f53e8424663ea6848a"), "x" : 1 }
> db.test.insert({"x" : 2});
unauthorized
> db.auth("test_user", "efgh");
1
> db.test.insert({"x": 2});
> db.test.find();
{ "_id" : ObjectId("4bb007f53e8424663ea6848a"), "x" : 1 }
{ "_id" : ObjectId("4bb0088cbe17157d7b9cac07"), "x" : 2 }
> show dbs
ssert: assert failed : listDatabases failed:{
   "assertion" : "unauthorized for db [admin] lock type: 1
",
   "errmsg" : "db assertion failure",
   "ok" : 0
}
> use admin
switched to db admin
> db.auth("root", "abcd");
1
> show dbs
admin
local
test
```

在建立连接之初，无法在名为 test 的数据库中进行任何读写操作。以用户 `read_user` 的身份通过验证后，可运行简单的 `find` 指令。尝试写入数据时，却因没有权限而再次操作失败。用户 `test_user` 在创建时并没有被设定为只读用户，因此可正常进行读写操作。但用户 `test_user` 并非管理员用户，因此不能通过执行 `show dbs` 指令来列出所有数据库。以上操作中的最后一步是管理员用户 `root` 的身份验证，`root` 可对任一数据库进行操作。

18.1.2　配置身份验证

启用身份验证后，客户端必须登录才能进行读写。然而，在 MongoDB 中有一点值得注意：在 admin 数据库中建立用户前，服务器上的”本地“客户端可对数据库进行读写。

一般情况下这不是问题，正常新建管理员用户并进行身份验证即可。唯一的例外情况则与分片有关。分片时，数据库 admin 会被保存在配置服务器（config server）上，所以分片中的 mongod 甚至并不知道它的存在。因此，在它们看来，它们虽然开启了身份验证但却不存在管理员用户。于是，分片中会允许一个本地的（local）客户端无需身份验证便可读写数据。

希望这不会成为一个问题，将网络配置为只允许客户端访问 mongos 进程即可。不过，如担心客户端在分片的本地上运行，不通过 mongos 进程而直接连接到分片的话，可在分片中添加管理员用户。

注意，我们并不想让分片集群知道这些管理员用户的存在，因为已经存在了一个 admin 数据库。在分片上建立的 admin 数据库仅供我们使用。要进行这一操作，可连接到每个分片的主节点，然后运行 `addUser()` 函数：

```
> db.addUser("someUser", "theirPassword")
```

应保证新建用户的副本集是作为集群中的分片存在的。如果新建了管理员用户，并尝试使用 `addShard` 命令将 mongod 作为分片加入集群，会发现这一命令无法执行（因为集群中已经存在了名为 admin 的数据库）。

18.1.3　身份验证的工作原理

数据库中的用户是作为文档被储存在其 `syste.users` 集合中的。这种用以保存用户信息的文档结构是 `{user : `*`username`*`, readOnly : `*`true`*`, pwd : `*`password hash`*`}`。其中 `password hash` 是基于 `username` 和密码生成的散列值。

了解了用户身份信息的存储位置与方法后，可方便地对其进行管理。例如，要删除一个用户，只需从 system.users 集合中删除这一用户的文档即可。

```
> db.auth("test_user", "efgh");
1
> db.system.users.remove({"user" : "test_user"});
> db.auth("test_user", "efgh");
0
```

用户进行身份验证时，服务器可通过绑定执行 `authenticate` 命令的连接，跟踪身份验证。这意味着只要驱动程序或其他工具使用了连接池或遇到故障而切换到另一节点，已经过身份验证的用户也需在新的连接上重新进行认证。这一操作应由驱动程序在后台进行处理。

18.2 建立和删除索引

本书第 5 章介绍了用于建立索引的命令，但没有深入介绍这些命令的运行过程。建立索引是数据库最耗费资源的操作之一，所以应小心地安排建立索引。

建立索引需 MongoDB 查找集合中每一个文档内被索引的字段（或正要建立索引的字段），然后对查找到的值进行排序。不出所料，随着集合体积的增长，该操作消耗非常大。因此，建立索引时，应使用对生产服务器影响最小的方式。

18.2.1 在独立的服务器上建立索引

在独立的服务器上，可在空闲时间于后台建立索引。除此之外，没有什么更好的办法来减轻建立索引所需的性能开销。在后台建立索引，可利用 `background：true` 参数运行 `ensureIndex` 命令：

```
> db.foo.ensureIndex({"somefield" : 1}, {"background" : true})
```

任何类型的索引均可在后台完成建立。

在前台建立索引要比在后台建立索引耗时少，但在索引建立期间会锁定数据库，从而导致其他操作无法进行数据读写。而后台在建立索引期间，则会定期释放写锁，从而保证其他操作的运行。这意味着后台建立索引耗时更长，尤其是在频繁进行写入的服务器上。但后台服务器在建立索引期间，可继续为其他客户端提供服务。

18.2.2 在副本集上建立索引

在副本集上建立索引最简单的方式，即在主节点中建立索引，然后等待其被复制到其他备份节点。在较小的集合中，这一操作不会造成太大的影响。

如集合较大，则会出现所有备份节点同时开始建立索引的情况。突然间所有备份节点都无法被客户端读取了，同时可能也无法及时进行同步复制。因此，对于较大的集合，推荐采用的方式是：

(1) 关闭一个备份节点；
(2) 将其作为独立的节点启动，如第 6 章描述的那样；
(3) 在这一服务器上建立索引；
(4) 重新将其作为成员加入副本集；
(5) 对每个备份节点执行同样的操作。

完成以上操作后，只剩主节点还没有建立索引。现在有两种选择。

- 于后台在主节点中建立索引（这会对主节点的性能造成压力）；
- 关闭主节点，并执行以上步骤 (1)~(4)，像在备份节点中一样，在主节点上建立索引。该方式需数据库停运一次，应权衡利弊进行选择。

也可以使用这种隔离创建技术，在没有被配置为建立索引的副本集内的成员上建立索引，即使用了 `buildIndexes：false` 选项。方法是将其作为独立服务器启动，建立索引，并重新加入副本集。

如果由于某种原因无法使用以上方法，则需计划在空闲时间（晚上、假期、周末等）来建立新的索引。

18.2.3　在分片集群上建立索引

在分片集群上建立索引，与在副本集中建立索引的步骤相同，不过需要在每个分片上分别建立一次。

首先，关闭均衡器。然后按照上一节中的步骤，依次在每一个分片中进行操作，即把每个分片当作一个单独的副本集。最后，通过 mongos 运行 `ensureIndex`，并重新启动均衡器。

只有在现存分片中添加索引时才需这样做，新的分片会在开始接收集合数据块时抓取集合中的索引。

18.2.4　删除索引

如不再需要某索引，可使用 `dropIndexes` 命令并指定索引名来删除索引。查询 system.indexes 集合找出索引名，即使是自动生成的索引名，在不同驱动器间也会存在些许差异：

```
> db.runCommand({"dropIndexes" : "foo", "index" : "alphabet"})
```

只需将 `"*"` 作为 `index` 的值，即可删除一个集合中的所有索引：

```
> db.runCommand({"dropIndexes" : "foo", "index" : "*"})
```

但这种方法无法删除 `_id` 索引。只有删除整个集合才能删除掉该索引。删除集合中的全部文档不会对索引产生影响，新文档插入后索引仍可正常增加。

18.2.5　注意内存溢出杀手

Linux 的内存溢出杀手（OOM Killer，out-of-memory killer）负责终止使用过多内存

的进程。考虑到 MongoDB 使用内存的方式，除了在建立索引的情况下，它通常不会遇到这种问题。如在建立索引时，mongod 突然消失，请检查 /var/log/messages 文件，其中记录了 OOM Killer 的输出信息。在后台建立索引或增加交换（swap）空间可避免此类情况。如拥有机器的管理员权限，可将 MongoDB 设置为不可被 OOM Killer 终止。更多信息请参见 23.5.2 节。

18.3 预热数据

重启机器或启动一台新的服务器，会耗费一段时间供 MongoDB 将所有所需数据从磁盘中载入内存。如对于性能的需求很高，要求数据必须出自内存中，则将新服务器上线，并等待应用程序载入所有所需数据，这会是一项艰巨的工作。

有几种方式可在服务器正式上线之前将数据载入内存，以避免在应用运行时带来麻烦。



重启 MongoDB 会改变内存中的内容。内存是由操作系统进行管理的，而操作系统不会将数据清除出内存，除非有其他程序需要使用此段内存空间。因此，如果 mongod 进程需要重启，应不会影响内存中的数据。（然而，mongod 会报告较低的常驻内存值，直到它有机会向操作系统请求所需的数据。）

18.3.1 将数据库移至内存

如需将数据库移至内存中，可使用 UNIX 中的 `dd` 工具，从而在启动 mongod 前载入数据文件：

```
$ for file in /data/db/brains.*
> do
> dd if=$file of=/dev/null
> done
```

将 `brains` 替换为需载入的数据库名。

将 `/data/db/brains.*` 替换为 `/data/db/*` 可将整个数据目录（即所有数据库）载入内存（假设内存容量足够的话）。如将一个或一组数据库载入内存，需占用的内存又要比实际内存大的话，则其中一些数据会立即被清除出内存。在这种情况下，可使用下一节中讲到的几种方法，以而将特定的数据载入内存中。

mongod 启动时，会向操作系统请求数据文件。如果操作系统发现内存中已经存在了这些数据文件，就可以立即访问此 mongod。

然而，只有在整个数据库可以装入内存中时，这一技术才能发挥作用。否则，可使用以下介绍的技术，来进行更多细粒度的（fine-grained）预热。

18.3.2　将集合移至内存

MongoDB 提供了 touch 命令来预热数据。启动 mongod（也许在另一个端口上，或关闭防火墙对它的限制），对一个集合使用 touch 命令，从而将其载入内存：

```
> db.runCommand({"touch" : "logs", "data" : true, "index" : true})
```

这一操作会将 logs 集合中的所有文档和索引载入内存。可指定内存只载入文档或只载入索引。touch 操作结束后，可允许应用访问 MongoDB。

然而，一整个集合（即使只有索引）依然可占用很大的空间。例如，应用可能只需要一个索引或一小部分文档在内存中。在这种情况下，需对数据进行自定义预热。

18.3.3　自定义预热

如需进行更复杂的预热，可自定义预热脚本。以下是一些常见的预热需求和解决方案。

- *加载一个特定的索引*

  假设索引必须处于内存中，如 {"friends" : 1, "date" : 1}。可进行覆盖查询（covered query，参见 5.1.2 节），从而将该索引载入内存中：

  ```
  > db.users.find({}, {"_id" : 0, "friends" : 1, "date" : 1}).
  ... hint({"friends" : 1, "date" : 1}).explain()
  ```

  以上 explain 命令会强制 mongod 遍历所有结果。必须通过 find 命令的第二个参数指定只返回被索引字段，否则这一查询同样会将所有文档加载入内存（也许这就是我们想要的结果，但应注意这一点）。注意，该操作总是会把无法被覆盖（covered）的文档和索引加载入内存，如多值索引（multikey index）。

- *加载最近更新的文档*

  如在更新文档时同时更新了日期字段上的索引，可通过查询最近日期来加载最近更新的文档。

  如没有日期字段上的索引，查询后会将集合中的所有文档加载入内存，所以就不必使用此方法了。在缺少日期字段的情况下，如主要关心的是最近插入的文档，可使用 _id 字段作为替代（参见下列内容）。

- 加载最近创建的文档

  如 `_id` 字段使用 `ObjectIdsIf` 类型，则可利用最近创建文档内的时间戳进行文档查询。如希望查找上星期建立的所有文档，可建立一个比所有要查找的文档建立时间都要早的 `_id`：

  ```
  > lastWeek = (new Date(year, month, day)).getTime()/1000
  1348113600
  ```

  将 `year`、`month` 和 `date` 进行适当替换，返回的结果是以秒为单位的日期值。现在需要使用此日期建立一个 `ObjectId` 类型的值。首先，将其转换成一个十六进制字符串，然后在后面加上 16 个 0：

  ```
  > hexSecs = lastWeek.toString(16)
  505a94c0
  > minId = ObjectId(hexSecs+"0000000000000000")
  ObjectId("505a94c00000000000000000")
  ```

  现在只需对其进行查询：

  ```
  > db.logs.find({"_id" : {"$gt" : minId}}).explain()
  ```

  该操作会加载自上星期以来的所有文档（以及 `_id` 索引的一部分右子树）。

- 重放应用使用记录

  MongoDB 提供有名为诊断日志（diaglog，diagnostic log）的功能来记录和回放操作流水。启用诊断日志会造成性能损失，所以最好通过临时使用的方式来获得一份“有代表性”的操作流水。在 mongo shell 中运行以下命令来记录操作流水：

  ```
  > db.adminCommand({"diagLogging" : 2})
  ```

  其中参数值为 2 意味着记录读取操作。该值为 1 时会记录写入操作，为 3 时读写都会进行记录（默认值为 0，意味着不进行记录）。我们可能不希望记录写入操作，因为在重放操作流水时，该操作会导致新成员产生额外写入。

  现在让 mongod 运行所需的时间并向其发送请求，从而令诊断日志记录操作流水。读取操作会被存放在诊断日志生成的文件中，该文件位于数据目录下。完成后将 `diagLogging` 的值重设为 0：

  ```
  > db.adminCommand({"diagLogging" : 0})
  ```

  要想使用诊断文件，可从该文件所在的服务器启动新的服务器，运行以下命令：

  ```
  $ nc hostname 27017 < /data/db/diaglog* | hexdump -c
  ```

按需对其中的 IP 地址、端口和数据目录进行替换。以上命令会将诊断文件中记录的操作作为普通请求发送到 hostname:27017 处。

注意，诊断日志会记录开启诊断日志的命令，所以，重放完成后，需登录服务器并关闭诊断日志（我们可能也想删除从重放中生成的诊断文件）。

这些技术可结合起来使用。例如，可在重放诊断记录的同时加载若干索引；如果没有遇到磁盘 IO 瓶颈的话，也可以同时进行这些操作；或者也可以通过多个 shell 或者 `startParallelShell`（启动并行 shell）命令（如果 shell 在 mongod 本地的话）来进行操作：

```
> p1 = startParallelShell("db.Ǹnd({}, {x:1}).hint({x:1}).explain()", port)
> p2 = startParallelShell("db.Ǹnd({}, {y:1}).hint({y:1}).explain()", port)
> p3 = startParallelShell("db.Ǹnd({}, {z:1}).hint({z:1}).explain()", port)
```

将 `port` 替换为 mongod 所在的端口值。

18.4 压缩数据

MongoDB 会占用大量的磁盘空间。有时，大量数据被删除或更新后，会在集合中产生碎片。如数据文件中有很多空闲空间，但由于这些独立的空闲区块过小，从而使得 MongoDB 无法对其进行重新利用时，就产生了碎片。在这种情况下，会在日志中看到类似如下信息：

```
Fri Oct 7 06:15:03 [conn2] info DFM::ǸndAll(): extent 0:3000 was empty,
    skipping ahead. ns:bar.foo
```

该信息本身是无害的。然而，这意味着某一整个区段（extent）中不包含任何文档。为消除空区段，并高效重整集合，可使用 `compact` 命令：

```
> db.runCommand({"compact" : "collName"})
```

压缩操作会消耗大量资源，不应在 mongod 向客户端提供服务时计划压缩操作。推荐做法类似于建立索引时的做法，即在每个备份节点中对数据执行压缩操作，然后关闭主节点，进行最后的压缩操作。

在一个备份节点中运行压缩操作，会使其进入恢复状态（recovering state），即它会对读取请求返回错误，亦无法作为一个同步源。压缩操作结束后，其状态会重新变为备份节点（secondary state）。

压缩操作会将文档尽可能地安排在一起，文档间的间隔参数默认为 1。如需更大的间隔参数，可使用额外的参数来指定它：

```
> db.runCommand({"compact" : "collName", "paddingFactor" : 1.5})
```

间隔参数最小为 1，最大为 4。对间隔参数的设定不会持续生效，只会影响压缩过程中 MongoDB 重新安排文档时的间隔。压缩完成后，间隔参数会重新返回之前的值。

压缩操作并不会减少集合占用的磁盘空间，该操作只是将所有文档都安排在集合的开始部分，这样当集合继续增大时就可以使用后面的空余部分。因此，压缩操作只是在磁盘空间不足时的临时措施，它不会减少 MongoDB 所使用的磁盘空间大小，但可使 MongoDB 不再需要分配新的空间。

可通过运行 `repair`（修复）命令来回收磁盘空间。`repair` 命令会对所有数据进行复制，所以必须拥有和当前数据文件一样大小的空余磁盘空间。这时常是个大问题，因为运行 `repair` 的最常见原因就是机器的磁盘空间不多了。然而，如能挂载其他磁盘，则可指定一个*修复路径*，即 `repair` 命令复制文件时所使用的目录（新安装的驱动）。

由于 `repair` 操作会完全复制所有数据，因此可随时中断该操作而不影响原始数据集。如在 `repair` 操作的过程中遇到问题，可删除 `repair` 产生的临时文件而不会影响到数据文件。

在启动 mongod 时使用 `--repair` 选项（如需要，还可使用 `--repairpath` 选项）来进行修复。

可以在 shell 中调用 `db.repairDatabase()` 来修复数据库。

18.5　移动集合

可使用 `renameCollection` 命令来重命名集合。这一命令无法在数据库间移动集合，但可更改集合名。无论重命名的集合大小，该操作都基本上是瞬间完成的。在繁忙的系统上，这一操作会耗费几秒钟，但在生产环境中运行可不用担心性能的消耗。

```
> db.sourceColl.renameCollection("newName")
```

执行这一命令时可传入第二个参数，从而决定当名为 `newName` 的集合已经存在时应如何处理。该参数为 `true` 时，会删除名为 `newName` 的集合；为 `false` 时，则会抛出错误。后者是这一参数的默认值。

要想在数据库间移动集合，必须进行转储（dump）和恢复（restore）操作，或手动复制集合中的文档（使用 `find` 命令，遍历集合中的所有文档，从而将其插入到新的数据库中）。

可使用 `cloneCollection` 命令将一个集合移动到另一个不同的 mongod 中。

```
> db.runCommand({"cloneCollection" : "collName", "from" : "hostname:27017"})
```

无法使用 `cloneCollection` 命令在 mongod 中移动集合，这一命令只能用于服务器间的集合移动。

18.6 预分配数据文件

如知道 mongod 具体需要哪些数据文件，可运行以下脚本，从而在应用上线前预分配数据文件。如能确定数据库和操作记录的大小，至少是一段时间以内的大小，这一方法则尤其有用。

```
#!/bin/bash

# 确保传入数据库名
if test $# -lt 2 || test $# -gt 3
then
    echo "$0 <db> <number-of-files>"
fi

db=$1
num=$2

for i in {0..$num}
do
      echo "Preallocating $db.$i"
      head -c 2146435072 /dev/zero > $db.$i
done
```

将以上代码存入一个文件中（比如说 preallocate 文件），并将文件设置为可执行文件。切换至数据目录，按需执行以下脚本，分配数据文件：

```
$ # create test.0-test.100
$ preallocate test 100
$
$ # create local.0-local.4
$ preallocate local 4
```

数据库启动后首次访问数据文件时，不能对其中的任何文件进行删除。例如，如分配数据文件 test.0~test.100，而启动数据库后却发现只需使用 test.0~test.20，这时我们不应删除 test.21~test.100 的文件。只要 MongoDB 意识到这些文件的存在，文件删除后则会导致 MongoDB 发生异常。

第 19 章 持久性

持久性（durability）是操作被提交后可持久保存在数据库中的保证。从完全没有保障到完全保证持久性，MongoDB 可高度配置与持久性相关的设定。本章内容包括：

- MongoDB 如何保证持久性；
- 如何配置应用和服务器，从而获得所需的持久性级别；
- 运行时关闭日记系统（journaling）可能带来的问题；
- MongoDB 不能保证的事项。

如磁盘和软件运行正常，则 MongoDB 能够在系统崩溃或强制关闭后，确保数据的完整性。

注意，关系型数据库通常使用持久性一词来描述数据库事务（transaction）的持久保存。由于 MongoDB 并不支持事务，因此该词义在这里有些许不同。

19.1 日记系统的用途

MongoDB 会在进行写入时建立一条日志（journal），日记中包含了此次写入操作具体更改的磁盘地址和字节。因此，一旦服务器突然停机，可在启动时对日记进行重放（replay），从而重新执行那些停机前没能够刷新（flush）到磁盘的写入操作。

数据文件默认每 60 秒刷新到磁盘一次，因此日记文件只需记录约 60 秒的写入数据。日记系统为此预先分配了若干个空文件，这些文件存放在 /data/db/journal 目录中，文件名为 _j.0、_j.1 等。

长时间运行 MongoDB 后，日记目录中会出现类似 _j.6217、_j.6218 和 _j.6219 的文件。这些是当前的日记文件。文件名中的数字会随着 MongoDB 运行时间的增长而增大。数据库正常关闭后，日记文件则会被清除（因为正常关闭后就不再需要这些文件了）。

如发生系统崩溃，或使用 `kill -9` 命令强制终止数据库的运行，mongod 会在启动时重放日记文件，同时会显示出大量的校验信息。这些信息冗长且难懂，但其存在说明一切都在正常运行。可在开发时运行 `kill -9` 命令来终止 mongod 进程并重新启动，这样在生产环境中，如果发生相同状况，也会明白此时显示哪些信息是正常的。

19.1.1 批量提交写入操作

MongoDB 默认每隔 100 毫秒，或是写入数据达到若干兆字节时，便会将这些操作写入日记。这意味着 MongoDB 会成批量地提交更改，即每次写入不会立即刷新到磁盘。不过在默认设置下，系统发生崩溃时，不可能丢失间隔超过 100 毫秒的写入数据。

然而，对于一些应用而言，这一保障还不够牢固，因此可通过若干方式来获得更强有力的持久性保证。可通过向 `getLastError` 传递 `j` 选项，来确保写入操作的成功。`getLastError` 会等待前一次写入操作写入到日记中，而日记在下一批操作写入前，只会等待 30 毫秒（而非 100 毫秒）：

```
> db.foo.insert({"x" : 1})
> db.runCommand({"getLastError" : 1, "j" : true})
> // The {"x" : 1} document is now safely on disk（文档现已安全保存在磁盘上）
```

注意，这意味着如果在每次写入操作中都使用了 `"j" : true` 选项，则写入速度实际上会被限制为每秒 33 次：

（1 次 /30 毫秒）×（1000 毫秒 / 秒）= 33.3 次 / 秒

通常将数据刷新到磁盘并不会耗费这么长时间，所以如果允许 MongoDB 对大部分数据进行批量写入而非每次都单独提交，数据库的性能则会更好。然而，重要的写入操作还是会经常选用此选项。

提交一次写入操作，会同时提交这之前的所有写入操作。因此，如果有 50 个重要的写入操作，可使用“普通的”`getLastError`（不包括 `j` 选项），而在最后一次写入后使用含有 `j` 选项的 `getLastError`。如果成功的话，就可知道所有 50 次写入操作都已安全刷新到磁盘上。

如果写入操作含有很多连接，可通过并发写入，来减少使用 j 选项所带来的速度开销。此种做法可增加数据吞吐量，但也会增加延迟。

19.1.2　设定提交时间间隔

另一个减少日记被干扰几率的选项是，调整两次提交间的时间间隔。运行 setParameter 命令，设定 journalCommitInterval 的值（最小为 2 毫秒，最大为 500 毫秒）。以下命令使得 MongoDB 每隔 10 毫秒便将数据提交到日记中一次：

```
> db.adminCommand({"setParameter" : 1, "journalCommitInterval" : 10})
```

也可使用命令行选项 --journalCommitInterval 来设定这一值。

无论时间间隔设置为多少，使用带有 "j" : true 的 getLastError 命令都会将该值减少到原来的三分之一。

如客户端的写入速度超过了日记的刷新速度，mongod 则会限制写入操作，直到日记完成到磁盘的写入。这是 mongod 会限制写入的唯一情况。

19.2　关闭日记系统

对于所有生产环境的部署，都推荐使用日记系统，但有时我们可能需要关闭该系统。即使不附带 j 选项，日记系统也会影响 MongoDB 的写入速度。如果写入数据的价值不及写入速度降低带来的损失，我们可能就会想要禁用日记系统。

禁用日记系统的缺陷在于，MongoDB 无法保证发生崩溃后数据的完整性。在没有日记系统的前提下，一旦发生崩溃，那么数据肯定会遭到损坏，从而需要对数据进行修复或替换。这种情况下遭到损坏的数据不应继续投入使用，除非我们不在乎数据库会突然停止工作。

如果希望数据库在崩溃后能够继续工作，有以下几种做法。

19.2.1　替换数据文件

这是最佳选择。删除数据目录中的所有文件，然后获取新文件：可从备份中恢复，使用确保正确的数据库快照，如果是副本集成员的话，也可对其进行初始化同步。如果是一个数据量较小的副本集，重新同步可能是最好的选择，即先停止此成员的运行（如果它还没有停止运行的话），删除数据目录中的所有内容，然后重新启动它。

19.2.2 修复数据文件

如果既没有备份和复制，也没有副本集中的其他成员，则需抢救所有可能的数据。需对数据库使用一个”修复“工具，修复实质上是删除所有受损数据，不过可能不会留有太多完好的数据。

mongod 自带了两种修复工具，一种是 mongod 内置的，另一种是 mongodump 内置的。mongodump 的修复更加接近底层，可能会找到更多的数据，但耗时要更长（而另一种自带的修复方式也不见得很快）。另外，如使用 mongodump 的修复，在准备再次启动前，依然需要恢复数据的操作。

因此，应根据愿意在数据恢复中消耗的时间长短来进行决定。

要使用 mongod 内置的修复工具，需附带 `--repair` 选项运行 mongod：

```
$ mongod --dbpath /path/to/corrupt/data --repair
```

进行修复时，MongoDB 不会开启 27017 端口的监听，但我们可通过查看日志（log）的方式得知它正在做什么。注意，修复过程会对数据进行一份完整的复制，所以如有 80 GB 的数据，则需 80 GB 的空闲磁盘空间。为尽量解决这一问题，修复工具提供了 `--repairpath` 选项。这一选项允许在主磁盘空间不足时挂载一个“紧急驱动器”，并使用它来进行修复操作。`--repairpath` 选项的用法如下：

```
$ mongod --dbpath /path/to/corrupt/data \
> --repair --repairpath /media/external-hd/data/db
```

如果修复过程被强行终止，或者出现故障（如磁盘空间不足），至少不会使情况变得更糟。修复工具将所有的输出都写入新的文件中，不会对原有文件进行修改。因此原始数据文件不会比开始修复时变得更糟。

另一个选择是使用 mongodump 的 `--repair` 选项，就像这样：

```
$ mongodump --repair
```

这些选择都不是特别好，但它们应该可以让 mongod 重新运行在一个干净的数据集上。

19.2.3 关于mongod.lock文件

数据目录中有一个名为 mongod.lock 的特殊文件。该文件在关闭日记系统运行时十分重要（如启用了日记系统，则这一文件不会出现）。

当 mongod 正常退出时，会清除 mongod.lock 文件，这样在启动时，mongod 就会得知上一次是正常退出的。相反，如果该文件没被清除，mongod 就会得知上一次是异常退出的。

如果 mongod 监测到上一次是异常退出的，则会禁止再启动，这样我们就会意识到一份干净数据副本的需求。然而，有些人意识到可通过删除 mongod.lock 文件来启动 mongod。请不要这么做。通常，在启动时删除这一文件，意味着我们不知道也不在乎数据是否受损。除非如此，否则请不要这么做。如果 mongod.lock 文件阻止了 mongod 的启动，请对数据进行修复，而非删除该文件。

19.2.4　隐蔽的异常退出

不要删除锁文件的另一重要原因在于，我们甚至可能意识不到这是一次异常退出。假设我们需要重启机器进行例行维护。初始化脚本应负责在服务器关闭之前关闭 mongod。初始化脚本通常会先尝试正常关闭程序，但如在若干秒后依然没有关闭的话，则会选择强行关闭。在一个繁忙的系统上，MongoDB 完全可能耗费 30 秒来结束运行，正常的初始化脚本不会等待它正常关闭。因此，异常退出的次数可能比我们知道的要多得多。

19.3　MongoDB无法保证的事项

在硬件或文件系统出现故障等情况下，MongoDB 无法保证操作的持久性。尤其是在硬盘发生损坏的情况下，MongoDB 根本无法保证数据安全。

另外，不同的硬件和软件对于持久性的保障可能有所不同。例如，一些破旧的硬盘会在写入操作还在列队中等待之际，便报告称写入成功。MongoDB 无法防止这一层次的误报，如果此时系统崩溃，数据就可能会发生丢失。

基本上，MongoDB 的安全性与其所基于的系统相同，MongoDB 无法避免硬件或文件系统导致的数据损坏。可使用副本应对系统问题。如果一台机器发生了故障，还有另一台在正常工作。

19.4　检验数据损坏

可使用 `validate` 命令，检验集合是否有损坏。如检验名为 foo 的集合，代码如下：

```
> db.foo.validate()
{
    "ns" : "test.foo",
```

```
        "firstExtent" : "0:2000 ns:test.foo",
        "lastExtent" : "1:3eae000 ns:test.foo",
        "extentCount" : 11,
        "datasize" : 75960008,
        "nrecords" : 1000000,
        "lastExtentSize" : 37625856,
        "padding" : 1,
        "firstExtentDetails" : {
            "loc" : "0:2000",
            "xnext" : "0:f000",
            "xprev" : "null",
            "nsdiag" : "test.foo",
            "size" : 8192,
            "firstRecord" : "0:20b0",
            "lastRecord" : "0:3fa0"
        },
        "deletedCount" : 9,
        "deletedSize" : 31974824,
        "nIndexes" : 2,
        "keysPerIndex" : {
            "test.foo.$_id_" : 1000000,
            "test.foo.$str_1" : 1000000
        },
        "valid" : true,
        "errors" : [ ],
        "warning" : "Some checks omitted for speed. use {full:true}
            option to do more thorough scan.",
         "ok" : 1
    }
```

需重点注意的是结尾附近的 `valid` 字段，字段值为 `true`。否则，输出内容中会包含找到的数据损坏细节。

输出中的大部分内容，是有关集合的内部结构信息，于调试而言没有太大用处。更多有关集合内部结构的内容，请参见附录 B。

- `firstExtent`（首区段）
  该集合首区段（extent）的磁盘偏移量（disk offset）。本例中位于文件 test.0 处，字节偏移量（byte offset）为 0x2000。

- `lastExtent`（尾区段）
  该集合尾区段的偏移量。本例中位于文件 test.1 处，字节偏移量为 0x3eae000。

- `extentCount`
  该集合所占区段数量。

- `lastExtentSize`
  最近分配区段的字节数量。区段大小随集合的增长而增长，最大可达 2 GB。

- `firstExtentDetails`

  描述集合中首区段的子对象。其中包含指向相邻两个区段的指针（`xnext` 和 `xprev`）、区段的大小（注意，它比尾区段要小得多，通常首区段是很小的），以及指向区段中第一条和最后一条记录（record）的指针。记录是真正承载着文档的结构。

- `deletedCount`

  该集合从存在至今，共删除的文档数目。

- `deletedSize`

  该集合中空闲列表（free list），即所有有效空余空间的大小。不仅包括被删除文档所占的空间，还包括已被预分配给该集合的空间。

`validate` 命令只适用于集合，而不适用于索引。因此我们通常无法判断索引是否被损坏，除非遍历检查一遍，即查询每个索引在集合中对应的文档。通过遍历得出的结果即可判断索引是否被损坏。

如果程序提示了非法的 BSON 对象（invalid BSONObj），一般说明数据损坏了。最糟糕的错误则是提到了 `pdfile` 的错误。`pdfile` 可以说是 MongoDB 的数据存储核心，源于 `pdfile` 的错误基本说明数据文件已经损坏了。

如果遇到了数据损坏，则可在日志中看到类似如下内容：

```
Tue Dec 20 01:12:09 [initandlisten] Assertion: 10334:
    Invalid BSONObj size: 285213831 (0x87040011)
    first element: _id: ObjectId('4e5efa454b4ae20fa6000013')
```

如果显示的第一个元素已经被废弃，就没什么可做的了。如果第一个元素还是可见的（如上例中的 `ObjectId`），也许可删除损坏文档。可尝试运行：

```
> db.remove({_id: ObjectId('4e5efa454b4ae20fa6000013')})
```

将其中的 `_id` 替换为日志中看到的对应 `_id`。注意，如果数据损坏影响的不只是该文档，则这种技术可能不会奏效。这种情况下，我们可能仍需对数据进行修复。

19.5 副本集中的持久性

第 10 章曾讨论过副本集中的投票问题，即一次对副本集的写入操作，在写入副本集中的大多数成员中之前，可能先会进行回滚（rollback）。可将与此相关的选项和之前提到的日记系统的选项结合起来使用：

```
> db.runCommand({"getLastError" : 1, "j" : true, "w" : "majority"})
```

进行这一操作后，可保证写入操作写入到了主节点和备份节点中，其中只有对主节点的写入可保证持久性。理论上来讲，在进行写入到记录到日记内的 100 毫秒时间内，多数的服务器同时崩溃也是有可能的，这种情况下数据库会回滚到当前主节点的状态。虽然这是一种极端情况，但也说明其并非是完美的。遗憾的是要解决这一问题并不简单，但目前 MongoDB 社区正尝试改变这一情况。

第六部分

服务器管理

第 20 章
启动和停止MongoDB

我们在第 2 章中学习了有关启动 MongoDB 的基本命令。本章我们将就生产环境中配置 MongoDB 的重要选项展开深入的学习，内容包括：

- 常用选项；
- 启动和停止 MongoDB；
- 安全相关选项；
- 使用日志时的注意事项。

20.1 从命令行启动

执行 mongod 程序即可启动 MongoDB 服务器，mongod 在启动时可使用许多可配置选项，在命令行中运行 `mongod --help` 可列出这些选项。下列选项十分常用，需着重注意。

- `--dbpath`

  使用此选项可指定一个目录为数据目录。其默认值为 /data/db/（在 Windows 中则为 MongoDB 可执行文件所在磁盘卷中的 \data\db 目录）。机器上的每个 mongod 进程都需要属于自己的数据目录，即若在同一机器上运行三个 mongod 实例，则需三个独立的数据目录。mongod 启动时，会在其数据目录中创建一个 mongod.lock 文件，以阻止其他 mongod 进程使用此数据目录。若尝试启动另一个使用相同数据目录的 MongoDB 服务器，则会出现错误提示：

  ```
  "Unable to acquire lock for lockfilepath: /data/db/mongod.lock."
  ```

- `--port`

  此选项用以指定服务器监听的端口号。mongod 默认占用 27017 端口，除其他 mongod 进程外，其余程序不会使用此端口。若要在同一机器上运行多个 mongod 进程，则需为它们指定不同的端口。若尝试在已被占用的端口启动 mongod，则会出现错误提示：

  ```
  "Address already in use for socket: 0.0.0.0:27017"
  ```

- `--fork`

  启用此选项以调用 `fork` 创建子进程，在后台运行 MongoDB。

  首次启动 mongod 而数据目录为空时，文件系统需几分钟时间分配数据库文件。预分配结束，mongod 可接收连接后，父进程才会继续运行。因此，`fork` 可能会发生挂起。可查看日志中的最新记录得知正在进行的操作。启用 `--fork` 选项时，必须同时启用 `--logpath` 选项。

- `--logpath`

  使用此选项，所有输出信息会被发送至指定文件，而非在命令行上输出。假设我们拥有该目录的写权限，若指定文件不存在，启用该选项后则会自动生成一个文件。若指定日志文件已存在，选项启用后则会覆盖掉该文件，并清除所有旧的日志条目。如需保留旧日志，除 `--logpath` 选项外，强烈建议使用 `--logappend` 选项。

- `--directoryperdb`

  启用该选项可将每个数据库存放在单独的目录中。我们可由此按需将不同的数据库挂载到不同的磁盘上。该选项一般用于将本地数据库或副本放置于单独的磁盘上，或在磁盘空间不足时将数据库移动至其他磁盘。也可将频繁操作的数据库挂载到速度较快的磁盘上，而将不常用的数据库放到较慢的磁盘上。总之该选项能使我们在今后更加灵活地操作数据库。

- `--config`

  额外加载配置文件，未在命令行中指定的选项将使用配置文件中的参数。该选项通常用于确保每次重新启动时的选项都是一样的。详细内容请参见 20.1.1 节。

例如，要在后台启动一个服务器，监听 5586 端口，并将所有输出信息发送至 mongodb.log 文件中，可运行如下命令：

```
$ ./mongod --port 5586 --fork --logpath mongodb.log --logappend
forked process: 45082
all output going to: mongodb.log
```

注意，mongod 可能在意识到自身启动前，便开始预配置日志文件。这时，直到预配置完成，`fork` 命令才会返回命令提示符。可查看 mongodb.log 文件（或重定向日志文件）末尾，观察这一操作过程。

首次安装启动 MongoDB 时，应查看一下日志。这一点很容易被忽视，尤其是使用初始化脚本来启动 MongoDB 的时候。但日志中常包含重要的警告信息，及时解决这些问题可预防随之而来的错误。如启动时没有出现任何警告，那么就一切就绪了。（启动时发出的警告信息会同时出现在 shell 里。）

如在启动时出现了警告信息，应把它们记录下来。MongoDB 会因以下问题发出警告：运行于 32 位的机器上（MongoDB 并非为 32 位机器设计）；启用了 NUMA（Non-Uniform Memory Access，非均匀访存模型，启用此会严重拖慢应用的运行速度）；或者系统所允许打开的文件描述符（descriptor）数目过少（MongoDB 需使用大量的文件描述符）。

重启数据库时，日志的前部不会发生更改，所以一旦了解了日志内容，就完全可以使用初始化脚本来运行 MongoDB，而不用去考虑日志。然而，在安装、升级，或从崩溃中恢复后，都应重新检查日志，以确保 MongoDB 和系统相契合。

启动数据库时，MongoDB 会将一个文档写入 local.startup_log 集合中，该集合包含了 MongoDB 的版本、其所基于的系统，以及所用的标记：

```
> db.startup_log.findOne()
{
    "_id" : "spock-1360621972547",
    "hostname" : "spock",
    "startTime" : ISODate("2013-02-11T22:32:52Z"),
    "startTimeLocal" : "Mon Feb 11 17:32:52.547",
    "cmdLine" : {
    },
    "pid" : 28243,
    "buildinfo" : {
        "version" : "2.4.0-rc1-pre-",
        ...
        "versionArray" : [
            2,
            4,
            0,
            -9
        ],
        "javascriptEngine" : "V8",
        "bits" : 64,
        "debug" : false,
        "maxBsonObjectSize" : 16777216
    }
}
```

该集合可用于跟踪数据库升级或更改后的运行状况。

使用配置文件

MongoDB 支持从文件中读取配置信息。当使用的选项很多，或自动化启动任务时，使用配置文件就十分实用。使用 `-f` 或 `--config` 标记，告知服务器使用配置文件。例如，运行 `mongod --config ~/.mongodb.conf`，从而使用 `~/.mongodb.conf` 作为配置文件。

配置文件中支持的参数和在命令行中的参数完全相同。以下是一个配置文件的例子：

```
# Start MongoDB as a daemon on port 5586

port = 5586
fork = true # daemonize it!
logpath = /var/log/mongodb.log
logappend = true
```

该配置文件指定的参数与之前启动时在命令行中指定的参数相同。文件中也展现了 MongoDB 配置文件的主要内容：

- `#` 后的内容，会被作为注释忽略掉；
- 指定参数的语法是 *`option = value`*，其中 `option` 的名称区分大小写；
- 在命令行中类似 `--fork` 的开关选项，应把 `fork` 的值设为 `true` 。

20.2 停止MongoDB

安全停止运行中的 MongoDB 服务器，与安全启动该服务器一样重要。有若干不同选项可有效地完成这一操作。

关闭运行中的服务器，最简洁的方法是使用 `shutdown` 命令——`{"shutdown" : 1}`。这是一个管理员命令，必须运行在 admin 数据库上。shell 提供了一个辅助函数，用以简单地执行 `shutdown` 命令：

```
> use admin
switched to db admin
> db.shutdownServer()
server should be down...
```

在主节点（primary）上运行 `shutdown` 命令时，服务器在关闭前，会先等待备份节点（secondary）追赶（catch up）主节点以保持同步。这将回滚的可能性降至最低，但 `shutdown` 操作有失败的可能性。如几秒钟内没有备份节点成功同步，则 `shutdown` 操作失败，主节点亦不会停止运行：

```
> db.shutdownServer()
{
    "closest" : NumberLong(1349465327),
    "difference" : NumberLong(20),
    "errmsg" : "no secondaries within 10 seconds of my optime",
    "ok" : 0
}
```

可使用 force 选项，强制关闭主节点：

```
db.adminCommand({"shutdown" : 1, "force" : true})
```

这相当于发送一个 SIGINT 或 SIGTERM 信号（三种做法都能使 MongoDB 安全地停止运行，但可能会有数据未能完成同步）。如服务器正在终端中作为前台进程运行，那么按下 Ctrl-C 快捷键也能发送一个 SIGINT 信号。另外，kill 之类的命令也可用于发送这些信号。假设 mongod 的 PID（Process identifier，进程标识符）为 10014，那么相应的命令就是 kill -2 10014（发送 SIGINT 信号）或 kill 10014（发送 SIGTERM 信号）。

mongod 收到 SIGINT 或 SIGTERM 信号后，会安全地停止运行。这意味着 mongod 会等当前正在进行的操作或文件预分配结束（耗时一定时间），再关闭所有打开的连接，将缓存写入磁盘，继而结束运行。

20.3 安全性

不要将 MongoDB 服务器直接暴露在外网上。应尽可能地限制外部对 MongoDB 的访问。最好的方式是设置防火墙，只允许内部网络地址对 MongoDB 的访问。第 23 章介绍了 MongoDB 服务器与客户端间的必要连接。

除使用防火墙外，也可在配置文件中加入以下选项来增强安全性。

- --bind_ip
  指定 MongoDB 监听的接口。我们通常将其设置为一个内部 IP 地址，从而保证应用服务器和集群中其他成员的访问，同时拒绝外网的访问。如 MongoDB 与应用服务器运行于同一台机器上，则可将其设为 localhost。但配置服务器和分片需要其他机器的访问，所以不应设为 localhost。

- --nohttpinterface
  MongoDB 启动时，默认在端口 1000 启动一个微型的 HTTP 服务器。该服务器可提供一些系统信息，但这些信息均可在其他地方找到。对于一个可能只需通过 SSH 访问的机器，没有必要将这些信息暴露在外网上。

除非正在进行开发，否则请关闭此选项。

- `--nounixsocket`
  如不打算使用 UNIX socket 来进行连接，则可禁用此选项。只有在本地，即应用服务器和 MongoDB 运行在同一台机器上时，才能使用 socket 进行连接。

- `--noscripting`
  该选项完全禁止服务器端 JavaScript 脚本的运行。大多数报告的 MongoDB 安全问题都与 JavaScript 有关。如程序允许的话，禁止 JavaScript 通常会更安全一些。

  一些 shell 中的辅助函数依赖于服务器端的 JavaScript，尤其是 `sh.status()`。在一台禁止了 JavaScript 的服务器上运行这些辅助函数时，会出现错误提示。

不要启用 REST 操作界面。该界面是默认禁用的，开启后可在服务器上执行很多命令，但并非为生产环境所设计。

20.3.1 数据加密

截止至撰写本书时，MongoDB 还未提供内置数据加密机制。如需对数据进行加密，可使用加密文件系统。另一种做法是手动加密某些字段，但 MongoDB 无法查询加密的值。

20.3.2 SSL安全连接

连接至 MongoDB 传输的数据默认不被加密。然而，MongoDB 支持使用 SSL 连接。由于授权的原因，默认版本中并未包含 SSL，可从 http://www.10gen.com 下载一个支持 SSL 的版本。也可以自己编译 MongoDB 的源代码启用 SSL。请查阅本国语言的驱动程序文档，了解创建 SSL 连接的方法。

20.4 日志

mongod 默认将日志发送至 stdout（标准输出，通常为终端）。大多初始化脚本会使用 `--logpath` 选项，将日志发送至文件。如在同一台机器上有多个 MongoDB 实例（比如说一个 mongod 和一个 mongos），注意保证各实例的日志分别存放在单独的文件中。确保知道日志的存放位置，并拥有文件的读访问权限。

MongoDB 会输出大量日志消息，但请不要使用 `--quiet` 选项（该选项会隐藏部分日志消息）。保持日志级别为默认值通常不错，此时日志中有足够的信息进行基本调试（如耗时过长或启动异常的原因等），但日志占用的空间并不大。调试应用某特定问题时，可使用一些选项从日志中获取更多信息。

首先，在重启 MongoDB 时，可通过在参数中附加数目更多的“v”（即 -v、-vv、-vvv、-vvvv 或 -vvvvv），或运行如下 `setParameter` 命令，完成日志级别（log level）的更改。

```
> db.adminCommand({"setParameter" : 1, "logLevel" : 3})
```

记得将日志级别重设为 0，否则日志中会存在过多不必要的内容。可将日志级别调高至 5，这时 mongod 会在日志中记录几乎所有的操作，包括每一个请求所处理的内容。由于 mongod 将所有内容都写入了日志文件，因此可产生大量的磁盘读写操作（IO），从而拖慢一个忙碌的系统。如需即时看到正在进行的所有操作，打开分析器不失为更好的方法：

MongoDB 默认记录耗时超过 100 毫秒的查询信息。如 100 毫秒不适用于应用，可通过 `setProfilingLevel` 命令来更改此阈值：

```
> // 只记录耗时超过 500 毫秒的查询操作
> db.setProfilingLevel(1, 500)
{ "was" : 0, "slowms" : 100, "ok" : 1 }
> db.setProfilingLevel(0)
{ "was" : 1, "slowms" : 500, "ok" : 1 }
```

上述第二条指令将关闭分析器，但第一条指令中以毫秒为单位的值将继续作为所有数据库中日志记录的阈值而生效。也可使用 `--slowms` 选项重启 MongoDB 来更改这一阈值。

最后，设置一个计划任务以便每天或每周分割（rotate）日志文件。如使用 `--logpath` 选项启动 MongoDB，向进程发送一个 SIGUSR1 信号即使其对日志进行分割。也可使用 `logRotate` 命令以达到相同目的：

```
> db.adminCommand({"logRotate" : 1})
```

如不是通过 `--logpath` 选项启动的 MongoDB，则不能对日志进行分割。

第 21 章

监控MongoDB

在部署前设置某种监控系统很是重要。监控系统应该能够跟踪服务器正在运行的操作，也能够在遇到问题时及时发出警报。本章将学习：

- 如何跟踪监测 MongoDB 的内存使用状况；
- 如何跟踪监测应用的性能指标；
- 如何诊断复制中的问题。

本章以 MMS（Mongo Monitoring Service，Mongo 监控服务）为例，演示监控时应注意的内容。请于 https://mms.10gen.com 查找 MMS 的安装说明。如不想使用 MMS，也可使用其他监控系统。监控系统可在故障发生前检测到潜在问题，并有助于我们对于问题的诊断。

21.1　监控内存使用状况

访问内存中的数据很快，而访问磁盘中的数据则较慢。不幸的是，二者相比，内存较为昂贵，而 MongoDB 通常也会优先使用内存。本节将讲述有关 MongoDB 与内存和磁盘进行交互的监控方式，以及监控中应关注的内容。

21.1.1　有关电脑内存的介绍

电脑中一般会有容量小且访问速度快的内存，以及容量大但访问速度慢的磁盘。当请求一页存储于磁盘上但尚未存于内存中的数据时，系统就会产生一个缺页中断，而后将此页数据从磁盘复制到内存。此后就可以极快地访问内存中的页面。如程序

不再使用此页面内容，而内存又被其他页所占满，旧的页面就会被清除出内存而只存在于磁盘上。

将一页数据从磁盘复制到内存，比从内存中读取一页数据耗时更长。因此，MongoDB 从磁盘复制数据的操作越少越好。如果 MongoDB 能够在内存中进行几乎所有操作，则访问数据的速度就能快很多。所以，MongoDB 的内存使用情况，是要跟踪监测的最重要指标之一。

21.1.2　跟踪监测内存使用状况

MongoDB 所使用的内存有几种不同的类型。第一种是*常驻内存*（resident memory）：MongoDB 在物理内存中明确拥有的内存部分。例如，在查询文档时，该页面即被载入内存中，并成为常驻内存的一部分。

MongoDB 赋予页面一个地址，此地址并非物理内存中页面的真实地址，而是一个*虚拟地址*。MongoDB 可将此地址传给内核，继而由内核将其翻译成真正的物理地址。这样，即使内核需将此页面从内存中清除出去，MongoDB 依然可通过虚拟地址来访问此页面。MongoDB 向内核请求内存，内核会在它的页缓存（page cache）中进行查找。但请注意，该页并不在此处。查找失败会产生缺页中断，继而将此页复制至内存，最后返回到 MongoDB。这些 MongoDB 赋予了地址的数据页面，即构成了*映射内存*（mapped memory），其中包含了 MongoDB 访问过的所有数据。通常情况下，映射内存的大小约等于整个数据集的大小。

MongoDB 为映射内存中的每个页面，都额外维护了一个虚拟地址，以供日记（journaling）使用（参见第 19 章）。这并不意味着内存中有着两份同样的数据，有的只是两个地址而已。所以，MongoDB 所使用*虚拟内存*的总量，约是映射内存的两倍大小，或者说是整个数据集的两倍大小。如关闭了日记机制，则映射内存的大小约等于虚拟内存的大小。

注意，虚拟内存和映射内存均不是“真正的”内存分配：二者与实际占用的内存大小毫无关系，它们只是 MongoDB 维护的映射罢了。理论上，MongoDB 可映射 1 PB（petabyte，1PB=1000 TB=1 000 000 GB）的内存，但实际只使用了几 GB 的物理内存。所以不用担心映射内存或虚拟内存的大小超过物理内存的容量。

图 21-1 是 MMS 中内存信息的图像，描述了 MongoDB 所使用的常驻内存、虚拟内存和映射内存的大小。在一台专门用于运行 MongoDB 的机器上，假设工作集（working set）的大小不小于内存容量，则常驻内存的大小应稍小于总的内存大小。只有常驻内存的大小才确切地等于其在物理内存中所占用的空间，但这一数据并不能说明 MongoDB 所使用的内存大小。

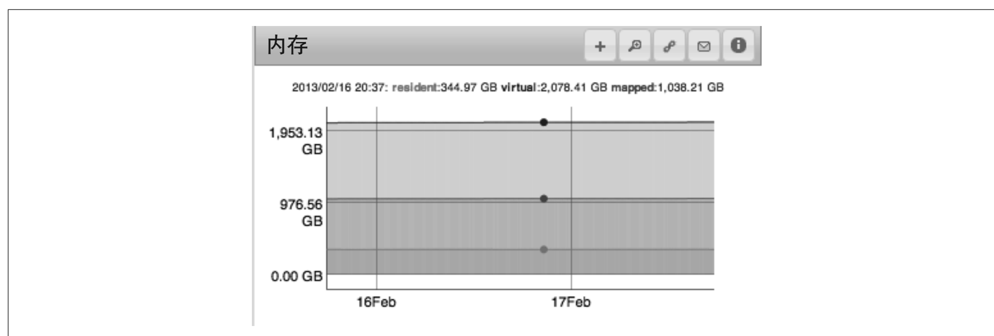


图 21-1：从上至下依次为虚拟内存、映射内存和常驻内存

数据如果能全部存放在内存中的话，则常驻内存应与数据差不多大小。当说到数据“在内存中”时，通常指的是在物理内存中。

21.1.3 跟踪监测缺页中断

如图 21-1 所示，内存的使用状况通常比较稳定，但随着数据集的增长，虚拟内存和映射内存也得到了增长。常驻内存会增长到可用物理内存的大小，而后保持不变。

除了以每种内存各占用多少空间为依据，还可通过其他数据得知 MongoDB 的内存使用方式。其中很实用的一个指标是发生缺页中断的数量，该数量表明了 MongoDB 所寻找的数据不在物理内存中这一事件的发生频率。图 21-2 和图 21-3 为一段时间内发生缺页中断的次数。图 21-3 中缺页中断的发生次数较少，但这一数据本身并不能说明很多问题。如果图 21-2 中的磁盘能够处理这么多的缺页中断，而应用程序可以处理这些磁盘操作造成的延迟，那么有这么多缺页中断也没什么问题。另一方面，如果应用程序无法处理从磁盘中读取数据造成的延迟，则只能将所有数据存放到内存中，或者使用固态硬盘（solid state drive，SSD）。

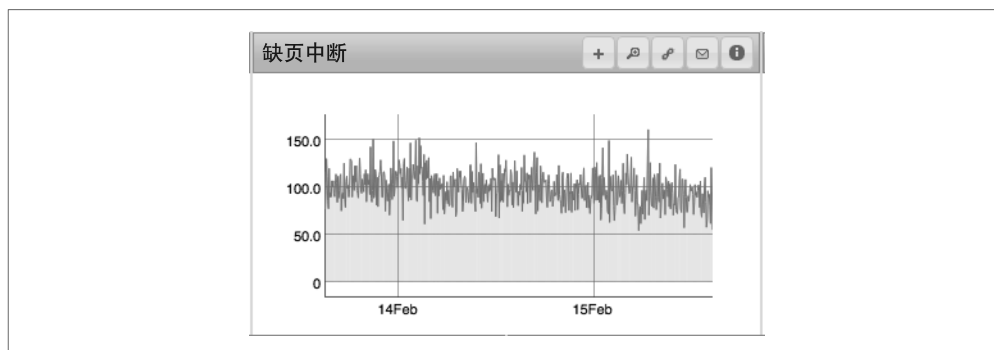


图 21-2：一个每分钟发生百余次缺页中断的系统

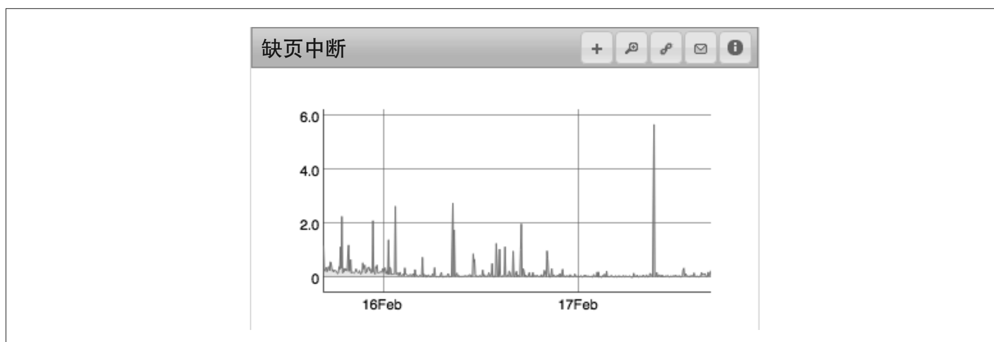


图 21-3：一个每分钟发生几次缺页中断的系统

无论应用能否处理这些延迟，缺页中断都会在磁盘超负荷时成为大问题。磁盘能够处理的读取操作数目并非是线性的：一旦磁盘开始超负荷运行，每个操作都必须排队等候更长时间，从而引发连锁反应。磁盘的超负荷运行通常存在一个临界点，超出临界点后磁盘的性能会迅速下降。因此，应避免磁盘的最大负荷运转。

监测一段时间内缺页中断的数量。如应用在某一数量的缺页中断下表现良好，则表明其为系统可处理的缺页中断数量底线。如应用性能在缺页中断上升至某一数值时开始发生恶化，则表明该数值是应发出警告的临界值。

在 `serverStatus` 命令输出的 `recordStats` 字段中，可看到每个数据库的缺页中断情况：

```
> db.adminCommand({"serverStatus" : 1})["recordStats"]
{
    "accessesNotInMemory": 200632,
    "test": {
         "accessesNotInMemory": 1,
         "pageFaultExceptionsThrown": 0
    },
    "pageFaultExceptionsThrown": 6633,
    "admin": {
        "accessesNotInMemory": 1247,
        "pageFaultExceptionsThrown": 1
    },
    "bat": {
        "accessesNotInMemory": 199373,
        "pageFaultExceptionsThrown": 6632
    },
    "config": {
        "accessesNotInMemory": 0,
        "pageFaultExceptionsThrown": 0
    },
    "local": {
        "accessesNotInMemory": 2,
```

```
            "pageFaultExceptionsThrown": 0
        }
    },
```

其中的 `accessesNotInMemory` 表示，MongoDB 自启动以来必须去磁盘上读取数据的次数。

21.1.4　减少索引树的脱靶次数

访问不在内存中的索引条目时效率尤其低下，因为这一操作通常会造成两次缺页中断，分别发生在将索引条目和文档加载入内存之际。查询索引造成缺页中断时，我们称其为*索引树的脱靶*（btree miss）。MongoDB 也会监测*索引树中靶*（btree hits）的次数，即无需到磁盘上访问索引。图 21-4 中可看到这两个数值。

索引十分常用，通常处于内存中，但如果内存过小而索引又过多，或访问模式不正常（例如进行大量的表扫描），都会造成索引树脱靶次数的增长。通常情况下，脱靶次数应保持在很小的数值，因此，一旦发现该数值过高，则须着手找出问题的源头。

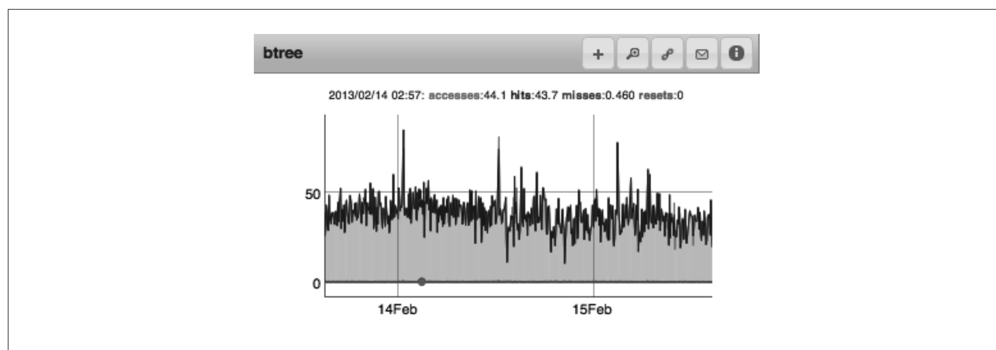


图 21-4：索引树状况图表

21.1.5　IO延迟

*IO 延迟*指 CPU 闲置等待磁盘响应的时间。通常情况下，该延迟与缺页中断密切相关。一些 IO 延迟是正常的，因为 MongoDB 有时须对磁盘进行访问，且无法完全避免对其他操作的妨碍。重要的是，需保证 IO 延迟不再持续增长或增至 100% 左右。如图 21-5 所示，这表明磁盘正在超负荷运转。

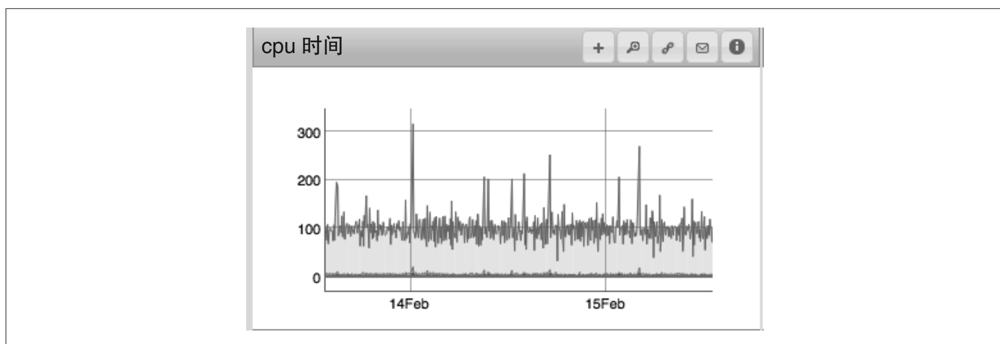


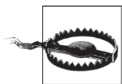
图 21-5：IO 延迟处于 100% 左右

MMS 可通过安装插件 munin 来监测 CPU 信息。如需查看安装说明，请访问 https://mms.10gen.com/help/install.html#hardware-monitoring-with-munin-node。

21.1.6　跟踪监测后台刷新平均时间

需关注的另一磁盘参数是，MongoDB 将脏页（dirty page）写入磁盘所花费的时间，即后台刷新平均时间（background flush average）。该数据相当于一个警钟。一旦所需时间开始延长，就表示磁盘的速度跟不上需要处理的请求。

MongoDB 默认会以至少每分钟一次的频率，将所有缓存中的数据刷新到磁盘中。（在有很多脏页的情况下，MongoDB 可能会以更高的频率进行刷新，这取决于操作系统。）可在启动 mongod 时，通过 `--syncdelay` 选项后的参数，以秒为单位，来配置这一时间间隔的值。同步的频率越高，每次同步的数据则更小，但效率也会随之降低。



人们常误以为 `syncdelay` 选项会影响数据持久性。但实际上二者毫无关系。要想确保数据持久性，应使用日记系统（journaling）。`syncdelay` 只用于调节磁盘性能。

通常情况下，我们希望后台刷新平均时间能够低于一秒。在繁忙的机器上或慢速磁盘上，该时间会有所延长，并且随着磁盘超负荷的运行，所需时间会变得越来越长。在某一时刻，磁盘超出负荷太多，以至于数据刷新用时超过 60 秒，这意味着 MongoDB 会不断尝试进行刷新（这又对磁盘造成了更大的负担）。磁盘刷新时间偶尔出现高峰是可以接受的。但不断出现数十秒的长时间写入则是我们不希望看到的。

图 21-6 为后台刷新平均时间的曲线变化图。该系统的硬盘驱动器压力很大，总是需要大于 5 秒的时间来写入前一分钟产生的数据。速度有些慢，尤其是经常会出现近

20 秒时长的高峰期，所以可能有必要将 `syncdelay` 的值调低一些，比如说 40 秒，然后看看每次刷新较少的数据是否会有帮助。

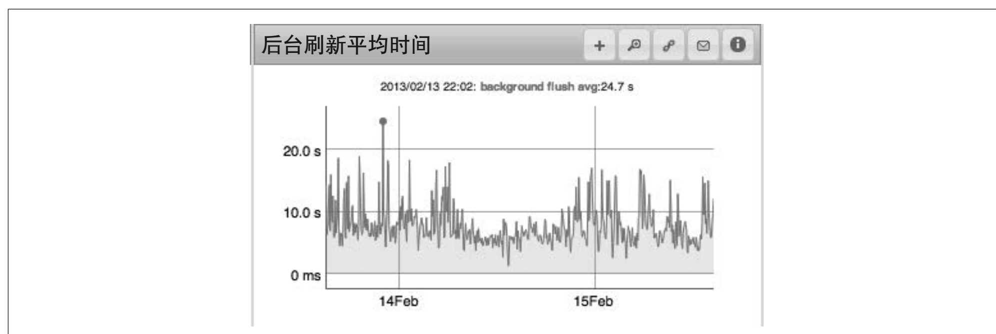


图 21-6：超负荷系统的后台刷新平均时间

如果后台刷新平均时间长时间超出磁盘所能承受的值（可能只超了几秒钟），就应该开始考虑如何减轻磁盘的负载。

MongoDB 只需刷新脏数据（即发生更改的数据），所以后台刷新平均时间通常反映出写入负载的大小，即写入操作和写入数据的数量。因此，如果写入负载很低，后台刷新平均时间可能无法表现出磁盘的压力大小。除后台刷新平均时间外，还应同时监测 IO 延迟和缺页中断的情况。

21.2 计算工作集的大小

通常情况下，内存中数据越多，MongoDB 的运行速度就越快。因此，应用可遇到如下情况（运行速度从快到慢排列）。

(1) 整个数据集均在内存中。虽然这种情况很不错，但通常代价过大或不可行。此种情况可能需要应用的响应速度足够快才能达成。

(2) 工作集处于内存中。这是最常见的选择。

工作集是应用所使用的数据和索引。这可能是其所有内容，但通常来讲会存在一个能够覆盖 90% 请求的核心数据集（如用户集合和最近一个月的活动）。如该工作集存在于物理内存中，MongoDB 的运行速度通常会很快，因为它只有在遇到少数“不寻常”的请求时才需访问磁盘。

(3) 索引处于内存中。

(4) 索引的工作集处于内存中。通常需要右平衡索引才能达成此种情况（详见第 5 章内容）。

(5) 内存中没有可用的数据子集。可能的话，应避免这种情况。这会使数据库运行缓慢。

我们必须通过了解工作集的内容及大小来判断能否将其存入内存。计算工作集大小的最好方式是跟踪分析一些常用的操作，从而找出应用的读写数据有多少。例如，假设应用每周会创建 2 GB 的新数据，而其中 800 MB 是经常被访问的。用户通常只会访问近一个月的数据，更早的数据则通常不会被用到。这样工作集大小可能是 3.2 GB（800MB/ 周 ×4 周）左右，再根据经验估计一下索引大小，加起来大概是 5 GB。

可通过跟踪监测一段时间内被访问的数据来考虑这一问题，如图 21-7 所示。如选择尽快满足 90% 的请求，则这一时间段内生成的数据和索引即为工作集，如图 21-8 所示。可测量这一时间的长短，从而计算出数据集的增长情况。注意，此例使用了时间，即数据的新旧作为参数，但同时可能存在更适用于应用的访问模式（时间是最常用的一种）。

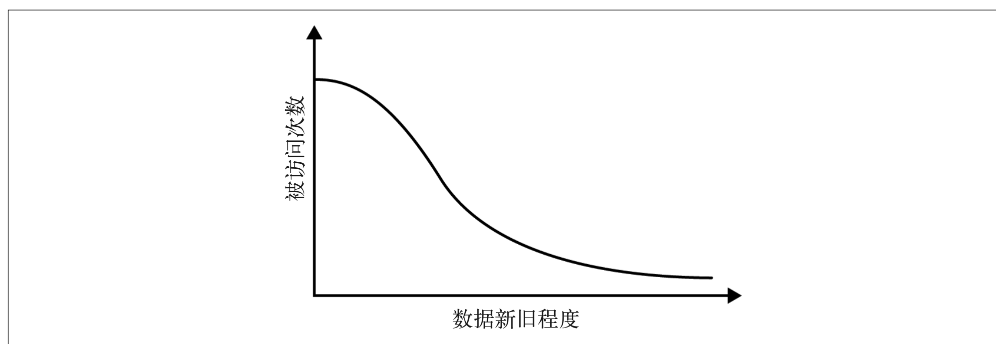


图 21-7：数据新旧程度与被访问次数的关系图

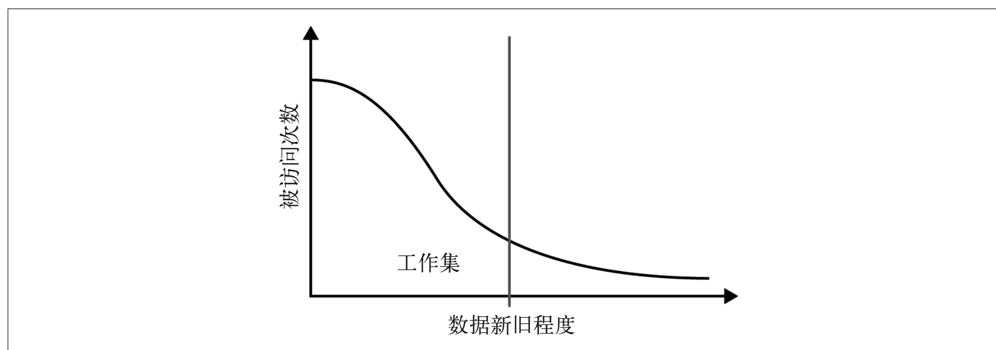


图 21-8：工作集即经常进行的请求所访问的数据

还可通过 MongoDB 的状态来估计工作集的大小。MongoDB 保留有一个记录内存内

容的图表，可将 `"workingSet" : 1` 参数传入 `serverStatus` 来得知这些内容：

```
> db.adminCommand({"serverStatus" : 1, "workingSet" : 1})
{
...
    "workingSet" : {
        "note" : "thisIsAnEstimate",
        "pagesInMemory" : 18,
        "computationTimeMicros" : 3685,
        "overSeconds" : 2363
    },
...
}
```

`pagesInMemory` 指 MongoDB 认为当前内存中的页面数目。实际上，MongoDB 并不知道其确切数值，但结果应该很接近。在返回信息中，如果内存中的页面数目与内存大小相等，则该数值没有什么价值；但如果页面数目小于内存大小，则该数值可能与工作集的大小有关。

`serverStatus` 的返回结果默认不包含 `workingSet` 字段。

一些工作集的例子

假设工作集大小为 40 GB。90% 的请求能够命中工作集，其他 10% 则需访问工作集以外的数据。如果有 500 GB 的数据和 50 GB 的内存，则工作集可全部放入内存中。一旦应用访问了需经常访问的数据（即预热过程），则无需在访问工作集时再次访问磁盘。有 10 GB 的空间提供给 460 GB 不常访问的数据。显然，MongoDB 几乎总是要到磁盘上访问工作集以外的数据。

另一方面，假设工作集无法放入内存。比如只有 35 GB 的内存。这种情况下工作集通常会占据大部分的内存。工作集中的内容经常被访问，因而更有可能留在内存中，但有时不常访问的数据也会被载入内存，从而将工作集（或其他不常访问的数据）挤出内存。于是，内存和磁盘会频繁进行数据交换，此时无法再预测访问工作集中数据的性能。

21.3　跟踪监测性能状况

查询的性能通常应重点监测并使其保持稳定。有几种方式可用来监测 MongoDB 是否能承受当前的请求负荷。

MongoDB 占用 CPU 时，大部分时间花在了处理器的读写上（IO 延迟很高，其他指标可忽略）。然而，如果用户或者系统占用的 CPU 时间接近 100%（或者 100% 乘

以 CPU 的数量），最可能的原因是一个常用的查询缺少合适的索引。另一种可能性是运行了太多的 MapRedues 或其他的服务器端 JavaScript 脚本。有必要跟踪监测 CPU，从而确保所有查询的表现与预想中的相符，特别是在部署了一个新版本的应用之后。

注意，图 21-9 中显示的是正常的，如果缺页中断的数量较低，IO 延迟可能被其他 CPU 活动所拖累。只有在其他活动增长时，缺少合适的索引才可能是罪魁祸首。

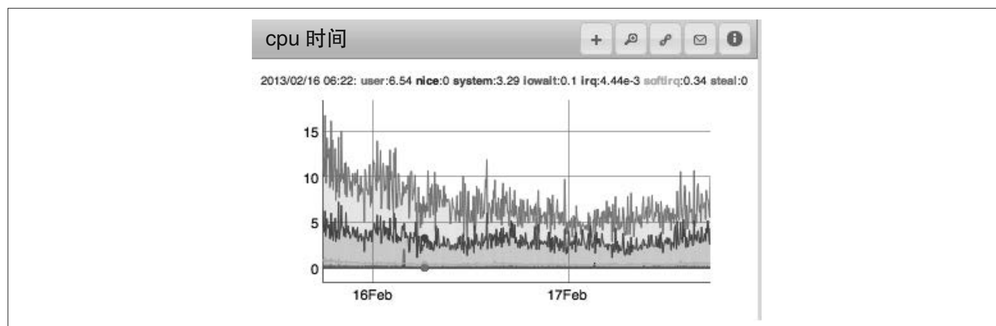


图 21-9：一个有着最小 IO 延迟的 CPU 状态图。上面的曲线表示用户的 CPU 时间，下面的是系统的 CPU 时间。其他数据都非常接近 0%

另一个相似的指标是队列长度，即有多少请求正在等待 MongoBD 的处理。请求在等待锁进行读写操作时，即被认为是处于队列中。图 21-10 为读写队列随时间变化的图像。不存在队列为最佳（此时图像基本为空白），但无需针对这一指标发出警报。在一个繁忙的系统中，操作需耗时等待以获取所需的锁，这一点很常见。

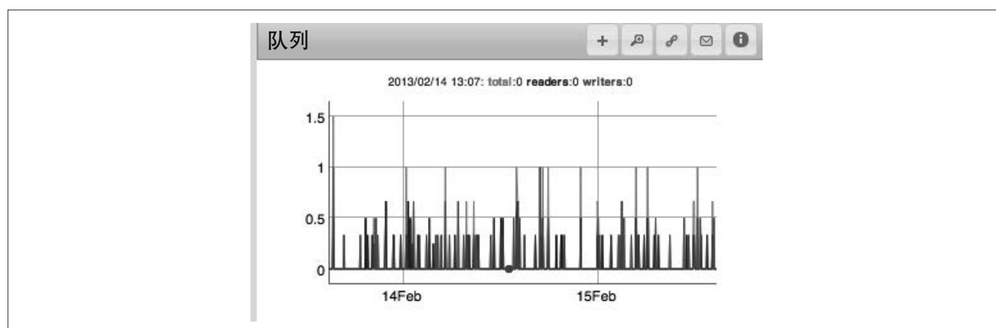


图 21-10：读写队列随时间变化的图像

可通过队列中的请求数量，判断是否发生了阻塞。通常队列的长度应该很低。一个很长且始终存在的队列表示 mongod 无法承受其负载。应尽快减轻该服务器的负荷。

可将队列长度和锁比例（lock percentage）两个指标结合起来，锁比例指 MongoDB 处于锁定中的时间。一般来讲，相较于发生锁定，磁盘 IO 更倾向于限制写入。但依然有必要对锁定进行跟踪监测，尤其是磁盘速度快，或连续写入多的系统。重复一遍，锁比例过高的最普遍原因之一就是缺少了合适的索引。随着锁比例的增加，操作取得锁所需的平均等待时间越来越长。因此，过高的锁比例会将所有东西拖慢，导致请求堆积，以及系统中更高的负荷和更高的锁比例。图 21-11 中显示了极高的锁比例，这种情况应尽快得到处理。

随着流量大小的变化，锁比例常会发生起伏变化。但如果锁比例长时间保持上升趋势，则表明系统所受的压力较大，应做一些调整。因此，应在锁比例长时间保持过高的值后再触发警报（这样当流量突然增加时就不会触发警报了）。

另一方面，我们可能也希望在锁比例突然升高时，比如说高于正常值 25% 时触发警报。该数值可能表明系统无法承载突然升高的负荷，也许应该提高系统的性能和容量了。

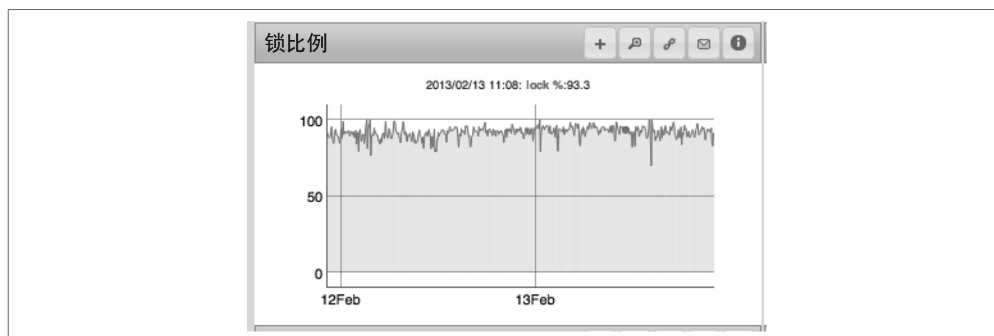


图 21-11：锁比例徘徊在 100% 附近，这种情况值得注意

除全局的锁比例外，MongoDB 也对每个数据库的锁比例进行跟踪。因此，如果某数据库有很多的连接，可单独查看其锁比例。

跟踪监测空余空间

另一基本但却很重要的监测指标为磁盘的使用情况，即监测磁盘的空余空间。有时用户直到磁盘空间被占满时才想起处理这一问题。通过监测磁盘使用情况，可预测当前磁盘的使用时间，并为磁盘空间不足提前做好准备。

磁盘空间不足时，有以下几个选项。

- 如果在使用分片，那就增加一个分片。

- 依次关闭副本集中的每个成员，复制数据到更大的磁盘上进行挂载。重启该成员，然后对下一成员进行同样的操作。
- 把副本集中的成员替换成更大驱动器的成员：移除旧成员，添加新成员。使新成员追赶上副本集中的其余成员。对集合中的每个成员重复此操作。
- 如使用了 directoryperdb 选项，且数据库增长速度非常快，可将数据库移至其驱动器内。挂载驱动器为数据目录。这样就可不必移动其他数据内容了。

无论采取哪种方法，请提前做好准备，从而使对应用产生的影响降至最低。请先做好备份，依次修改副本集中的每个成员，并将数据从一处复制至另一处。

21.4 监控副本集

对副本集中的落后（lag）和 oplog（operation log）长度进行跟踪监测十分重要。

当备份节点无法与主节点保持一致时，就产生了*落后*。主节点最后一次操作的时间和备份节点最后一次操作的时间差值，即落后的值。例如，一个备份节点刚刚完成了一次操作，其时间戳为 3:26:00 p.m.，主节点刚刚完成了一次操作，其时间戳为 3:29:45 p.m.，此时落后的值即为 3 分 45 秒。落后的值越接近 0 越好，且通常为毫秒级别。如果一个备份节点能够与主节点保持同步，副本集落后的值应如图 21-12 所示，基本保持为 0。

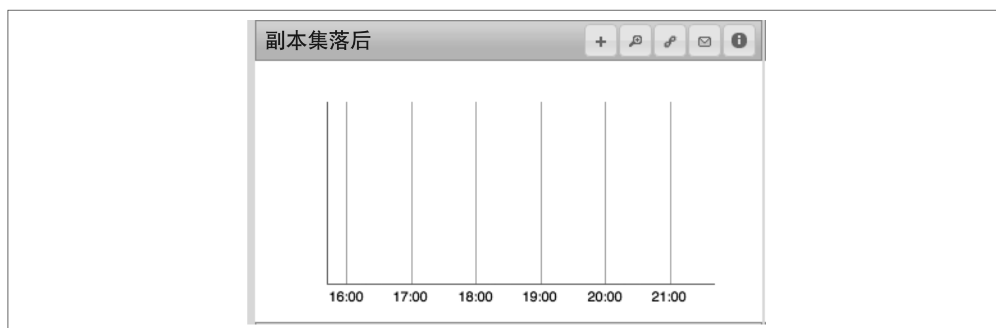


图 21-12：一个不存在落后的副本集，这是最理想的状态

如果备份节点的复制速度赶不上主节点的写入速度，就会开始出现非 0 的落后值。最极端的情况是副本集发生了*阻塞*：由于某种原因，副本集无法再接受任何操作。这种情况下，每经过一秒，落后的值就会增加一秒，在图像中呈现一个陡坡的样子，如图 21-13 所示。这可能是由于网络问题引起的，也可能是由于缺少了 _id 索引，副本集要求每个集合都拥有这一索引才能正常工作。

如果集合缺少了 _id 索引，将服务器脱离副本集作为一个独立服务器启动，然后建立 _id 索引。确保建立的 _id 索引是唯一索引（unique index）。索引建立完成后，

除非删除整个集合，否则 _id 索引不能发生删除或更改。

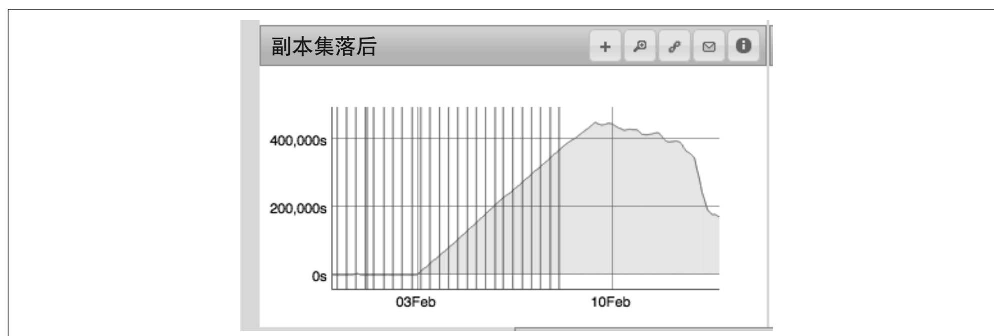


图 21-13：副本集发生阻塞，并于 2 月 10 日前开始进行恢复。红色线条表示服务器的重新启动

如系统超负荷运行，备份节点可能会逐渐被主节点落下。但图中通常不会显示出特征明显的“每秒增加一秒”的陡坡，因为备份节点还是进行了一些复制的。然而，备份节点到底是因为无法与高峰流量保持一致而被落下的，还是逐渐被主节点落下的，这一点十分重要。

主节点不会为了“帮助”备份节点追赶上来而限制写入，所以在超负荷运行的系统上备份节点追赶不上的情况时有发生（尤其是 MongoDB 中写入的优先级比读取要高，这意味着副本集的性能很大程度上取决于主节点）。可在写入时使用“w”参数来强制限制主节点的写入。也可通过将请求路由至其他成员，从而降低备份节点的负载。

而在一个负载极低的系统上，可在副本集落后值的图像中看到另一种有趣的图案，即突然出现的高峰值，如图 21-14 所示。这些峰值表示的并不是真正的落后，而是由抽样的变化产生的。mongod 每隔几分钟处理一个写入操作。落后的值是主节点和备份节点的时间戳差值，而对备份节点时间戳的测量恰好发生在主节点的写入操作之前，这使得备份节点看起来好像落后了几分钟一样。如果增加写入频率，这些峰值就会消失。

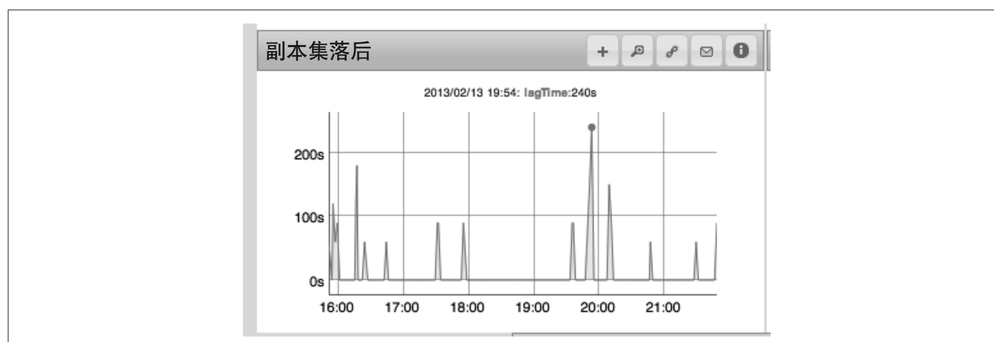


图 21-14：写入操作数量较少的系统会产生“伪落后”

另一需要跟踪监测的重要指标是每个成员的 oplog 长度。每个可能成为主节点的成员都应拥有一份长度超过一天的 oplog。如一个成员可能成为另一个成员的同步源（sync source），则应拥有一份长度足够进行初始化同步（initial sync）的 oplog。图 21-15 为标准的 oplog 长度图像。该长度极佳，达 1111 小时，即超过一个月的数据！通常，在保证磁盘空间充足的前提下，oplog 应尽可能地长。oplog 几乎不占用内存。而且长 oplog 的缺乏，可能会带来痛苦的回忆。

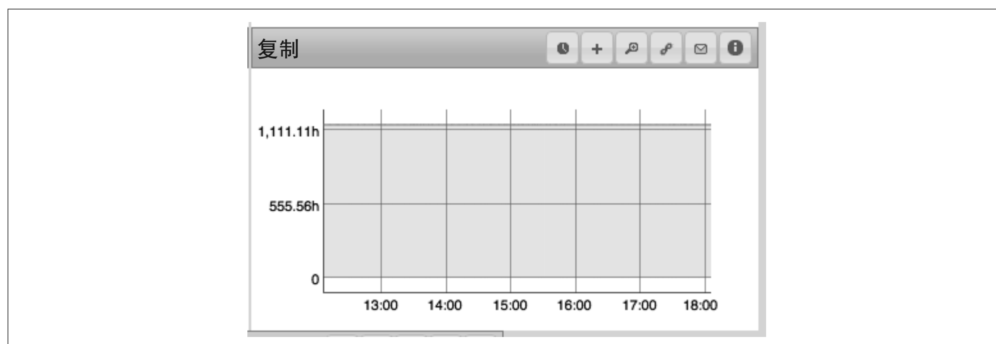


图 21-15：典型的 oplog 长度图像

图 21-16 为较短的 oplog 和变化的流量引起的稍显不同寻常的变化。运行仍旧正常，但该机器上的 oplog 可能太短了（6 到 11 小时的维护时段）。管理员有机会的话应将该 oplog 的长度加以延长。

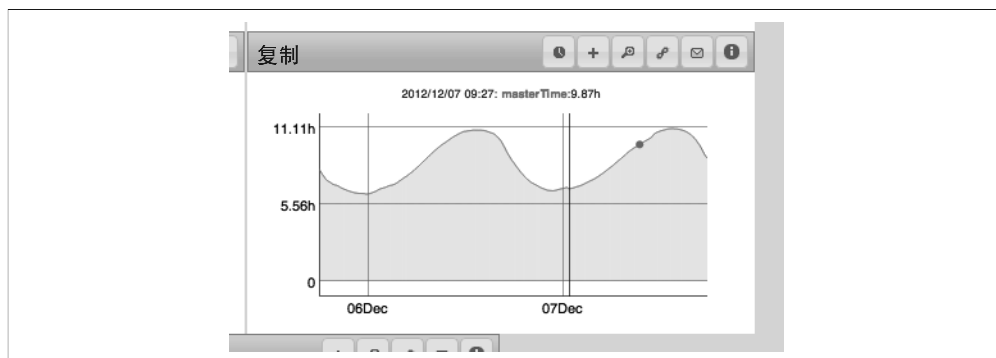


图 21-16：每天有一次流量高峰的应用 oplog 长度

第22章

备份

对系统进行定期备份是很重要的。对于大多故障而言，备份是很好的保护措施，只有很少的故障无法通过恢复干净的备份得到解决。本章我们将学习下列有关备份的常用选项：

- 单一服务器的备份；
- 对副本集进行备份时的特别考虑；
- 如何对一个分片集群进行备份。

只有在有信心能在紧急情况下完成迅速部署的情况下，备份才是有用的。所以，无论选择了哪种备份技术，一定要对备份及恢复备份的操作进行练习，直到了然于心。

22.1　对服务器进行备份

备份有许多种方法。但无论采用哪种方法，备份操作都会增加系统的负担：备份通常需将所有数据读取到内存中。因此，通常情况下，应对副本集的非主节点（与主节点相对）进行备份，或在空闲时段对独立服务器进行备份。

如非特殊声明，本节中的所有技术均适用于任何 mongod 程序，无论是独立服务器还是副本集成员。

22.1.1　文件系统快照

生成文件系统快照（snapshot）是最简单的备份方法。然而，该方法的实现需要两

点条件，即文件系统本身支持快照技术，以及在运行 mongod 时必须开启日记系统（journaling）。如系统满足这两点条件，则该方法无需其他准备，只需生成快照即可，时间不限。

在恢复时，确保 mongod 没有在运行。从快照恢复数据的确切命令取决于不同的文件系统，不过基本上就是恢复快照，然后启动 mongod 即可。如果是对正在运行的系统生成快照，那么快照中的数据内容本质上相当于使用 `kill -9` 命令强制终止 mongod 后的数据内容。因此，mongod 在启动时会对日志（journal）文件进行重放（replay），然后开始正常运行。

22.1.2　复制数据文件

另一种备份方式是复制数据目录中的所有文件。没有文件系统的支持，我们就无法同时复制所有文件，因此在进行备份时必须防止数据文件发生改变。可使用 `fsynclock` 命令做到这一点：

```
> db.fsyncLock()
```

该命令锁定（lock）数据库，禁止任何写入，并进行同步（fsync），即将所有脏页刷新至磁盘，以确保数据目录中的文件是最新的，且不会被更改。

一旦运行了这一命令，mongod 会将之后的所有写入操作加入队列等待，且在解锁前不会对这些写入操作进行处理。注意，这一命令会停止所有数据库的写入操作，而不只是已连接的那个数据库。

当 `fsynclock` 命令返回命令行后，复制数据目录中的所有文件到备份位置。在 Linux 中，可使用以下命令等：

```
$ cp -R /data/db/* /mnt/external-drive/backup
```

确保复制了数据目录中的每一个文件和文件夹到备份位置。漏掉文件或文件夹可能会损坏备份或使其不再可用。

数据复制完成后，解锁数据库，使其能够再次进行写入操作：

```
> db.fsyncUnlock()
```

数据库即开始正常处理写入操作。

注意，身份验证和 `fsynclock` 命令存在一些锁定问题。如果启用了身份验证，则在调用 `fsyncLock()` 和 `fsyncUnlock()` 期间不要关闭 shell。如果在这期间断开了连接，则可能无法进行重新连接，并不得不重启 mongod。`fsyncLock()` 的设定在

重启后不会保持生效，mongod 总是以非锁定模式启动。

除使用 `fsynclock` 外，还可关闭 mongod，复制文件，然后重启 mongod。关闭 mongod 会将所有更改立即刷新到磁盘，防止备份期间出现新的写入操作。

若要恢复数据目录备份，请保证 mongod 没有在运行，且所有待恢复的数据目录为空。将备份的数据文件复制到数据目录，然后启动 mongod。例如，下列命令会使用前面提及的命令恢复备份文件：

```
$ cp -R /mnt/external-drive/backup/* /data/db/
$ mongod -f mongod.conf
```

忽略那些有关复制部分数据目录的警告信息。只要知道要复制哪些文件，即可使用这种方式备份单独的数据库。例如，要备份名为 myDB 的数据库，只需复制所有名为 myDB.* 的文件，包括后缀名为 .ns 的文件。如使用了 `--directoryperdb` 选项，只需复制该数据库对应的整个数据目录。

可复制数据库对应的文件到数据目录，完成指定数据库的恢复。如需进行这种部分恢复，应确保数据库上一次是正常关闭的。如遇到崩溃或突然停机，不要尝试恢复一个单独的数据库，而应用备份文件替换整个数据目录，然后启动 mongod，从而允许日记文件进行重放。



不要同时使用 `fsyncLock` 和 mongodump。数据库被锁定也许会使得 mongodump 永远处于挂起状态，这取决于数据库正在进行的其他操作。

22.1.3 使用mongodump

最后一种备份方式是使用 mongodump。之所以最后提到它，是因为 mongodump 有些许缺点。它备份和恢复的速度较慢，在处理副本集时存在一些问题（参见 22.2 节）。然而它也存在以下优点：当想备份单独的数据库、集合甚至集合中的子集时 mongodump 是个很好的选择。

运行 `mongodump --help`，可看到 mongodump 具有很多选项。此处我们重点关注那些与备份相关的实用选项。

要备份所有数据库，只需运行 mongodump 即可。如果在同一台机器上运行 mongod 和 mongodump，只需指定 mongod 运行时占用的端口即可：

```
$ mongodump -p 31000
```

mongodump 会在当前目录建立一个*转储*（dump）目录，其中包含了一份所有数据

的倾卸。转储目录中的目录和子目录由数据库和集合构成。真正的数据存放在扩展名为 .bson 的文件里，其中以 BSON 格式依次存储了集合中的所有文档。可使用 MongoDB 自带的 bsondump 工具查看 .bson 文件。

使用 mongodump 时甚至无需服务器处于运行状态：可使用 `--dbpath` 选项来指定数据目录，mongodump 会使用指定的数据文件进行备份。

```
$ mongodump --dbpath /data/db
```

如果 mongod 正在运行，则不应使用 `--dbpath` 选项。

mongodump 存在一个问题，即它并非进行快照备份，也就是说在备份的过程中，系统可能会继续进行写入操作。于是可能出现，开始备份时 mongodump 先对数据库 A 进行转储，随后在 mongodump 正在对数据库 B 进行转储的同时，删除了数据库 A。然而 mongodump 已经对数据库 A 进行了转储，于是最终转储得到的结果，是一个在原服务器上并不存在的数据快照。

为避免这种情况的发生，如果运行 mongod 时使用了 `--replSet` 选项，则可使用 mongodump 的 `--oplog` 选项。这会将转储过程中服务器进行的所有操作记录下来，这样在恢复备份时就会重新执行这些操作。这样就可以得到源服务器上某一时间点的数据快照。

如果给 mongodump 一个副本集的连接字串（例如，`setName/seed1,seed2,seed3`），如果备份节点存在的话，它会自动选择一个备份节点进行转储。

恢复 mongodump 产生的备份，可使用 mongorestore 工具：

```
$ mongorestore -p 31000 --oplogReplay dump/
```

如果转储数据库时使用了 `--oplog` 参数，运行 mongorestore 时必须使用 `--oplogReplay` 选项，以得到某一时间点的快照。

如果在运行的服务器上进行数据替换，可使用 `--drop` 选项，以在恢复一个集合前先删除它。当然此选项并非必选项。

随着版本的变化，mongodump 和 mongorestore 命令的具体作用和用法发生了改变。为避免兼容性问题，应尽量使用同版本的 mongodump 和 mongorestore。可运行 `mongodump --version` 和 `mongorestore --version` 来查看各自的版本。

1. 使用 mongodump和mongorestore来转移集合和数据库

可从转储中恢复完全不同的数据库和集合。当在不同环境中使用不同的数据库名称

（例如，dev 和 prod），但集合的名称相同时，这一特性会很实用。

将一个扩展名为 .bson 的文件恢复为特定的数据库和集合，只需在命令行中指定恢复目标：

```
$ mongorestore --db newDb --collection someOtherColl dump/oldDB/oldColl.bson
```

2. 管理唯一索引带来的混乱

在任何集合中，如果存在除 _id 以外的其他唯一索引（unique index），则应考虑使用 mongodump 和 mongorestore 以外的备份方式。具体地说，唯一索引要求复制期间数据不发生可能破坏其唯一索引约束的改变。最安全的方法是先想办法“冻结”数据，然后使用前两节中提到的方法来进行备份。

如果决定使用 mongodump 和 mongorestore 进行备份，那么在恢复备份时，可能需要对数据进行一定的预处理。

22.2　对副本集进行备份

通常，应该对备份节点进行备份：这会为主节点减轻负担，也可以在不影响应用的情况下锁定备份节点（只要应用不向备份节点发送读取请求）。可使用之前提到过的三种方式中的任意一种，对副本集中的成员进行备份，但推荐使用文件系统快照或复制数据文件的方式。这两种方式在应用于副本集备份节点时无需做任何修改。

副本集启用后，使用 mongodump 进行备份就不那么简单了。首先，如果使用 mongodump，则必须在备份时使用 --oplog 选项，来得到一个基于某时间点的快照；否则备份的状态不会和任何其他集群成员的状态相吻合。在恢复时也必须创建一份 oplog，否则被恢复的成员就不知道应该同步到哪里。

要从 mongodump 生成的备份中，对副本集成员进行恢复，可将该成员作为一个单独的服务器启动，此时要使用一个空的数据目录。首先，像上一节中提到过的那样，使用 --oplogReplay 选项运行 mongorestore。现在它应该包含了一份完整的数据副本，但还需要一份 oplog。运行 createCollection 命令来建立 oplog：

```
> use local
> db.createCollection("oplog.rs", {"capped" : true, "size" : 10000000})
```

以字节为单位指定集合大小。可参见 12.4.6 节，了解更多与此相关的内容。

现在需要填充 oplog。最简单的方式是用备份中的 oplog.bson 文件来填充 local.oplog.rs 集合：

```
$ mongorestore -d local -c oplog.rs dump/oplog.bson
```

注意，这并不是对于 oplog 的转储文件（dump/local/oplog.rs.bson），而是进行转储期间发生的操作。一旦 mongorestore 完成，即可将服务器作为副本集成员重新启动。

22.3 对分片集群进行备份

不可能对正在运行的分片集群进行“完美地”备份，因为无法及时得到集群在某一时间点完整状态的快照。然而，通常情况下都会避开该限制，因为随着集群的增大，从备份中恢复整个集群的可能性越来越小。因此，在面对分片集群时，我们更关注分块的备份，即单独备份配置服务器和副本集。

在对分片集群进行备份和恢复操作之前，应先关闭均衡器。这是因为在过于混乱的环境中是无法得到一份前后一致的快照的。有关均衡器开启与关闭的操作说明，请参见 16.4 节。

22.3.1 备份和恢复整个集群

当集群很小或正在进行开发时，我们可能想要转储和恢复整个集群。要达到这一目的，应先关闭均衡器，然后通过 mongos 运行 mongodump。这会在 mongodump 所运行的机器上建立所有分片的备份。

要恢复此种备份，需运行 mongorestore 并连接到一个 mongos。

关闭均衡器后，可使用文件系统快照或复制数据目录的方式，备份配置服务器和每一个分片。然而不可避免的是，我们不可能在完全相同的时刻得到这些备份，这可能造成问题。另外，在打开均衡器时会进行数据合并，在分片中备份的某些数据可能会由此消失。

22.3.2 备份和恢复单独的分片

更多时候，只需恢复集群中的某个单独分片。如果不是很挑剔的话，可使用刚刚在前面提到过的单独服务器处理方法进行分片的备份恢复。

有一个问题要着重注意：假设在星期一对集群进行了备份。到了星期四，磁盘发生损坏，我们不得不恢复备份。然而，在这几天里，新的数据块可能移动到了这一分片上。而周一进行的分片备份中并不包含这些新增的数据块。也许我们能够使用配置服务器的备份，找到这些消失了的数据块在星期一时的位置，但这比只是恢复分片要困难得多。在大多数情况下，恢复分片，忽略那些消失的数据块，是更好的选择。

可直接连接到一个分片上来恢复备份，而不需要通过 mongos。

22.4 使用mongooplog进行增量备份

以上提及的备份方式，即使和上一次备份时相比，只发生了很小的更改，也都必须对所有数据进行一次完整的复制。如果数据和写入量有很大的关系，那么我们可能希望了解一下增量备份。

与每天或每周进行一次完整的数据复制不同，我们只需进行一次备份，然后使用 oplog 来备份这之后的所有操作。这种技术比之前提及的技术都要复杂，因此除非确实需要，否则应尽量选择其他技术。

这一技术需要两台运行 mongod 的机器，即机器 A 和机器 B。A 是主机器（可能是副本集中的备份节点），B 则用来进行备份：

(1) 记录下 A 的 oplog 中最近一次的操作时间（optime）：

```
> op = db.oplog.rs.find().sort({$natural: -1}).limit(1).next();
> start = op['ts']['t']/1000
```

把该数值记录在安全的地方——等下会用到它。

(2) 对数据进行备份，使用以上提及的任何一种方式，得到一份基于某时间点的备份。恢复备份至 B 上的数据目录。

(3) 定期添加 A 上的操作至 B，从而完成数据的复制。MongoDB 的发行版中自带了一个特殊的工具 mongooplog（读作 mon-goop-log），将这一操作变得简单。mongooplog 从一台服务器的 oplog 中复制数据，并将其中的操作应用在另一台服务器的数据集上。在 B 上运行：

```
$ mongooplog --from A --seconds 1234567
```

其中 `--seconds` 选项后跟的参数，应为第一步中计算出的 `start` 变量和当前时间的差值，再额外加上几秒（重复地重放操作也好过数据丢失）。

这使得备份更接近最新的数据。这种技术有些像是手动地同步一个备份节点，所以我们也许只是想在备份节点上使用延时复制以代替增量备份。

第 23 章

部署MongoDB

本章将会就部署生产服务器给出相关建议。具体来讲，包括以下几方面：

- 选购硬件、挑选设置方法；
- 使用虚拟化环境；
- 重要的内核与磁盘 IO 设定；
- 网络设置：哪些组件之间需要建立连接。

23.1 设计系统结构

通常，我们会希望对系统进行优化，以保证数据安全和存取速度。本节将探讨在选择磁盘、RAID（磁盘阵列）配置、CPU 等硬件以及基本软件组件的过程中，达成以上目标的最佳方法。

23.1.1 选择存储介质

如果只考虑性能，可按照以下顺序选择介质，从而进行数据存取：

- 内存；
- 固态磁盘；
- 机械磁盘。

可惜，大多情况下，由于预算有限或数据过多，无法将所有数据存入内存，而固态磁盘又过于昂贵。因此，标准的部署方案是使用较少的内存空间（具体大小取决于

总数据大小）和较大的机械磁盘空间。这种情况下需注意，工作集大小应小于内存容量，同时应做好在工作集增长时进行设备扩展的准备。

如果没有经费限制，那就去购买更多的内存或固态磁盘。

从内存中读取数据需几纳秒的时间（比如 100 纳秒）。相反地，从磁盘中读取数据需几毫秒的时间（比如 10 毫秒）。单看这两个数字很难想像出二者间的差距，但如果我们将它们按比例放大就会明白：如果访问内存耗时 1 秒钟，则访问磁盘需耗时超过 1 天的时间！

100 纳秒 ×10 000 000 = 1 秒

10 毫秒 ×10 000 000 = 1.16 天

这些只是近似的计算（磁盘可能略快或内存略慢），但差距的大小不会有太大差别。所以我们会想要尽量少地访问磁盘。

即使是更快的机械磁盘，也不会使磁盘读取时间缩短太多，所以没有必要花太多钱在这种磁盘上。更多的内存或固态磁盘效果会更好。

一个示例

图 23-1 至图 23-6 展示了固态磁盘的优势。这些图片中显示的，是一个在 8 月 8 日中午上线的新分片的情况。开始时仅在机械磁盘上部署了一个分片，随后又在固态磁盘上部署了一个新的分片，接下来两个分片同时运行。

如图 23-1 所示，机械磁盘的性能峰值可接近每秒 5000 次查询，但一般情况下只能做到每秒几百次查询。

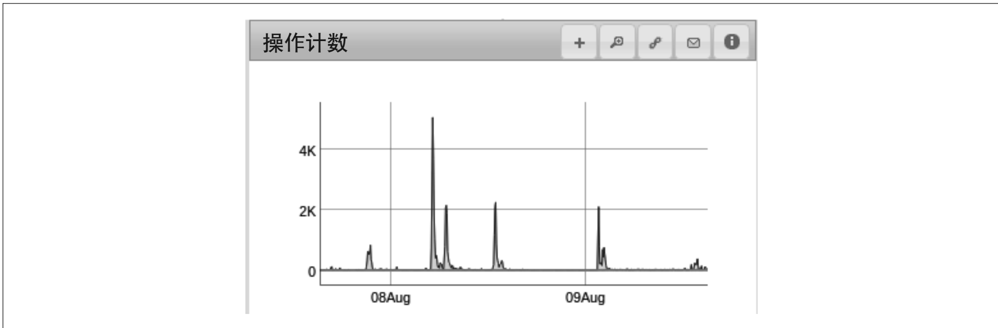


图 23-1：在机械磁盘上进行查询的情况

作为对照，图 23-2 中的图表显示了在固态磁盘上进行查询的状况。固态磁盘的性能可保持每秒处理 5000 次请求，峰值则可达到每秒 30000 次！这一新的分片完全可

以独立承担整个集群的工作。

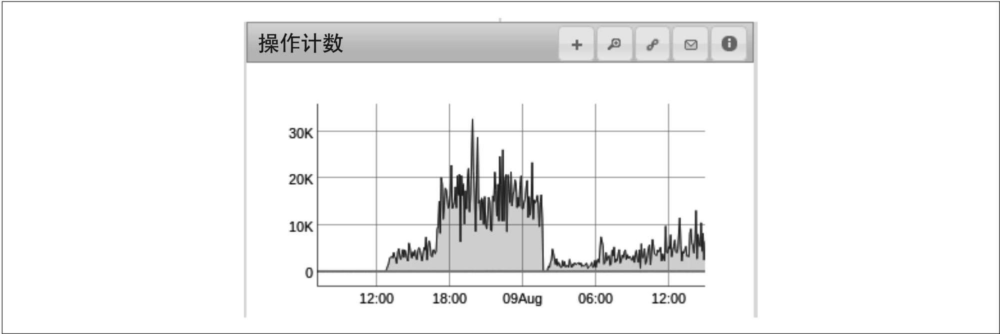


图 23-2：在固态磁盘上进行查询的情况

有关机械磁盘和固态磁盘的对比中，另一点值得注意的是频繁的磁盘访问对系统的压力大小。在使用机械磁盘的服务器上，我们可从其硬件监控信息（图 23-3）中看到，磁盘工作十分繁忙。图中位于上部的曲线表示 IO 延迟，即 CPU 等待磁盘 IO 的时间所占总时间的百分比。可以看到该百分比至少为 10%，高峰时常达到 50% 以上。这意味着磁盘成为了限制性能的短板（所以此人新添了固态磁盘）。

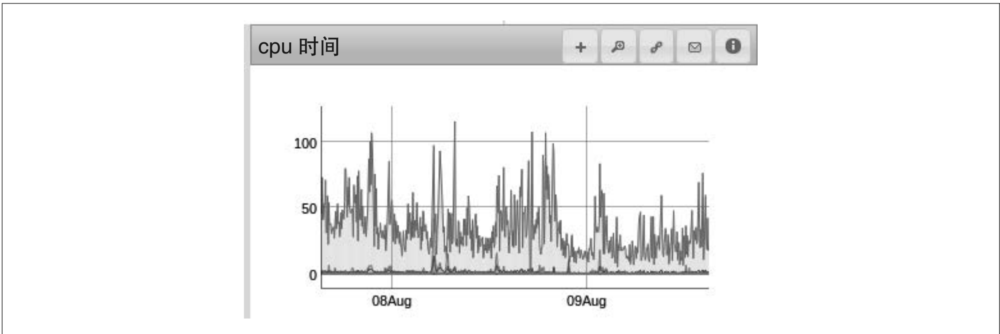


图 23-3：查询期间的 CPU 使用情况

作为对比，图 23-4 显示了使用固态磁盘的机器上 CPU 的使用情况。图中甚至已经看不出 IO 延迟的痕迹，上下两条明显的曲线分别表示系统时间（system time）和用户时间（user time）。因此，限制这一机器性能的短板就是 CPU 的运行速度。图中曲线超过了 100%，这也说明系统利用了多个处理器核心。将其与图 23-3 进行对比可发现，之前的机器由于磁盘 IO 速度过慢，导致得到充分利用的处理器核心甚至还不足一个。

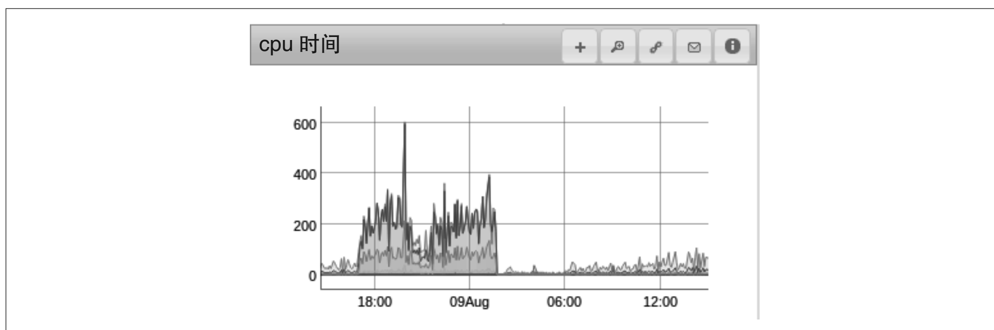


图 23-4：查询期间的 CPU 使用情况

最后，在有关锁时间的图 23-5 中可以看到其对 MongoDB 的影响。在机械磁盘上，数据库 10% 到 25% 的时间处在锁状态，有时峰值甚至会达到 100%。

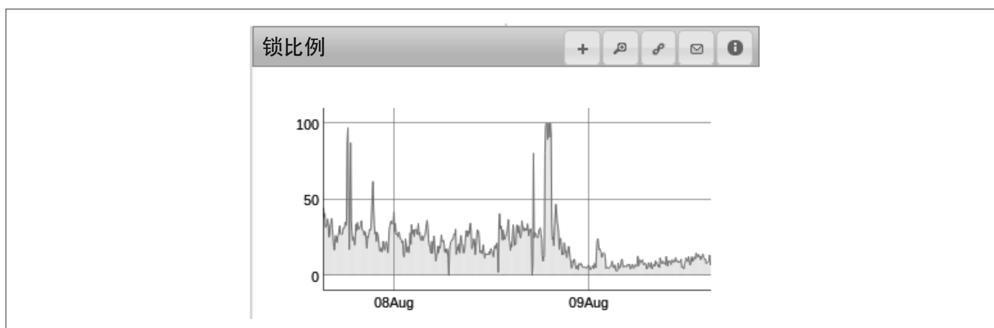


图 23-5：查询期间 MongoDB 的锁比例

与图 23-6 中使用固态磁盘机器上的锁比例进行比较。MongoDB 基本上一直保持非锁定状态。（曲线开始部分的凸起是在加上固态磁盘之前的数据读取操作造成的。）

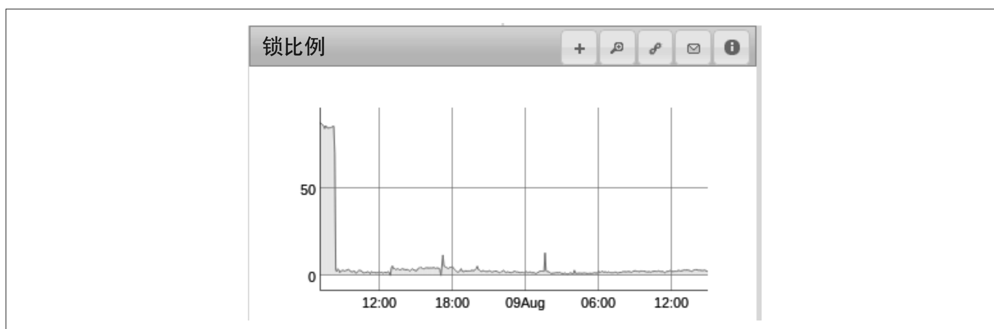


图 23-6：使用固态磁盘机器上的锁比例

可以看到，固态磁盘可以承担比机械磁盘多得多的工作，但不幸的是，它们无法被大量部署。如果能够使用它们，那就用吧。就算不可能在整个集群中使用固态磁盘，也应考虑尽量多得部署，然后使用在第 15 章中提到的强制热点数据模型进行优化。

注意，通常我们不能向已有的副本集中添加固态磁盘（如副本集中存在机械磁盘的话）。如果使用固态磁盘的机器成为主成员（primary member），并接管处理它所能处理的一切工作，则其他成员受速度所限，无法及时复制数据，从而被落在后面。因此，如果要引入固态磁盘的话，向集群中增加一个新的分片不失为一种更好的方法。

注意，固态磁盘对于处理普通数据而言表现优异，但实际上机械磁盘完全可以用于记录日志（journal）。用机械磁盘来记录日志，而用固态磁盘记录数据，这样既能节省固态磁盘的空间，也不会影响性能。

23.1.2　推荐的RAID配置

RAID（Redundant Array of Independent Disk，独立磁盘冗余阵列，旧称 Redundant Array of Inexpensive Disk，廉价磁盘冗余阵列）是一种可以让我们把多块磁盘当作单独一块磁盘来使用的技术。可使用它来提高磁盘的可靠性或性能，或二者兼有。一组使用 RAID 技术的磁盘被称作 RAID 磁盘阵列。

RAID 根据性能的不同，存在着多种配置方式，通常兼顾了速度与容错性。下列是几种最常见的配置方式。

- RAID0
  使用磁盘分割技术（disk striping）将多个磁盘并列起来以提升性能。每块磁盘保存一部分数据，与 MongoDB 中的分片类似。由于存在多个底层磁盘，因此大量数据可在同一时间写入磁盘内。这一方式可提高写入效率。然而，如果其中一块磁盘发生故障导致数据丢失，则这些数据不会存在备份。这也会导致读取速度变慢（尤其是在 Amazon 的 Elastic Block Store 服务上），因为一些数据卷可能比另一些要慢。

- RAID1
  使用镜像来提高可靠性。同样的数据副本会被写入到阵列的每一个成员当中。这一方法的性能要比 RAID0 低，因为阵列中一个速度慢的成员会拖慢整个阵列的写入速度。然而，如果其中一块磁盘发生故障，还可以在阵列中的其他成员上找到数据副本。

- RAID5
  在使用磁盘分割技术的基础上，额外存储数据的校验信息，以防服务器故障导致

数据丢失。一般情况下，在一块磁盘发生故障时 RAID5 可以自动处理它，用户并不会感觉到故障的发生。然而，这也使得 RAID5 成为这些 RAID 配置方案中最慢的一种，因为它需要在写入数据时计算校验信息。而 MongoDB 所进行的恰恰是典型的多次少量的数据写入工作，因此使用 RAID5 所带来的代价尤为可观。

- RAID10
  RAID10 是一种 RAID0 和 RAID1 的组合：数据被分割以提升速度，又被复制镜像以提高可靠性。

推荐使用 RAID10，它比 RAID0 更安全，也能解决 RAID1 的性能问题。有人觉得在副本集的基础上再使用 RAID1 有些浪费，从而选择 RAID0。这是个人喜好问题：你原意为了性能承担多大的风险呢？

不要使用 RAID5，它非常非常慢。

23.1.3　CPU

MongoDB 对于 CPU 的负载很轻（注意在图 23-3 和图 23-4 中：两个 CPU 的处理能力即可满足每秒 10 000 次查询）。如需在内存和 CPU 间选择一个进行硬件投资，一定要选择内存。理论上来讲，在进行读取或在内存中进行排序时，会耗尽多核的运算资源，但在实践中这种情况很少发生。在建立索引和进行 MapReduce（一个用于大规模数据集并行运算的软件架构）运算时，对 CPU 的负载很大，但直到本书写作之时，增加处理器核数仍无法对这两种操作起到优化作用。

如需在速度和核数间做出选择，应选择前者。相比更多的并行运算，MongoDB 能更好地利用单处理器上的更多周期进行运算。

23.1.4　选择操作系统

64 位 Linux 操作系统是运行 MongoDB 的最好选择。可能的话应选择它作为内核系统。CentOS 和 RedHat 企业版可能是最普遍的选择，其他的发行版也应能够运行 MongoDB（Ubuntu 和 Amazon Linux 也很常用）。应使用最新发布的稳定版本，因为老旧的、存在缺陷的软件包或内核有时会产生问题。

64 位 Windows 系统也能很好地运行 MongoDB。

MongoDB 对于其他版本 Unix 系统的支持并没有那么好：如果使用 Solaris 或者基于 BSD 的系统，那么应该小心，因为这些系统发布的 MongoDB，都存在（至少曾经存在）很多问题。

关于跨平台兼容，有一点需特别注意：MongoDB 在所有系统中使用同样的线路协议（wire protocol），对于数据文件中的内容也使用同样的格式进行存储，所以我们可以基于不同系统的组合来部署 MongoDB。例如，可在 Windows 系统上运行 mongos 进程，而在 Linux 上运行 mongods 来作为其分片。也可在 Windows 和 Linux 间复制数据文件，而不必考虑跨平台兼容的问题。

如服务器需处理大量数据，则不要使用 32 位系统，因为这会限制我们最多只能处理 2 GB 的数据（这是由于 MongoDB 使用内存映射的文件）。副本集的仲裁器和 mongos 进程可以运行在 32 位机器上。不要在 32 位机器上运行其他类型的 MongoDB 服务。

MongoDB 只支持小端（little-endian，即存储二进制内容时，数字的低位组置于最前面）系统结构。大部分驱动都支持小端和大端（big-endian）两种系统结构，因此客户端在两种系统中均可运行。然而，服务器只能运行在小端结构的机器上。

23.1.5　交换空间

应分配一小块交换空间，以防系统内存使用过多，从而导致内核终止 MongoDB 的运行。然而，MongoDB 通常并不会使用任何交换空间。

MongoDB 所使用的大部分内存都是“不稳定的”：只要系统因某些原因而请求内存空间，这部分内存中的内容就会被刷新到磁盘中，然后原内存则被替换成其他内容。因此，数据库数据绝不应该被写入交换空间，因为它首先会被刷新回磁盘。

然而，MongoDB 在需要对数据进行排序，即建立索引或进行排序操作时，会使用交换空间。在进行此类操作时，MongoDB 会尽量不去使用过多内存，但如果同时进行很多这种操作，最终就会使用到交换空间。

如果应用程序在服务器上用到了交换空间，则应想办法重新设计应用程序，或者减少那台服务器上的负载。

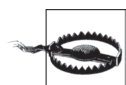
23.1.6　文件系统

在 Linux 系统上，推荐使用 ext4 或 XFS 文件系统作为数据卷。具有一个能够在备份时进行文件系统快照（filesystem snapshot）的文件系统是不错的，但是会影响到性能。

不推荐使用 ext3 文件系统，因为它在对数据文件进行预分配时耗时过长。MongoDB 会定期分配 2 GB 大小的数据文件并将其内容填充为 0。在 ext3 文件系统上，这一操

作会造成几分钟的卡顿。如果一定要使用 ext3 文件系统，有几个相关的优化措施可供选择。不过如果可以的话，还是应尽量使用其他文件系统。

在 Windows 系统上，使用 NTFS 和 FAT 文件系统都是可以的。



不要直接使用被挂载的 NFS 文件系统作为 MongoDB 的存储区域。有些版本的客户端会隐瞒数据刷新的真实情况，随机重新挂载和刷新页面缓存（page cache），且不支持排他文件锁定（exclusive file lock）。使用 NFS 文件系统会造成日志（journal）内容损坏，因此应尽量避免使用。

23.2 虚拟化

运用虚拟化（virtualization）技术可方便地使用廉价的硬件来部署系统，并且能够迅速做出扩展。然而，虚拟化也存在缺点，尤其是无法预知的网络和磁盘 IO 状况。本节将探讨有关虚拟化的具体问题。

23.2.1 禁止内存过度分配

内存过度分配（memory overcommitting）的设置值决定了当进程向操作系统请求过多内存时应采取的策略。基于这一设置，内核可能会为进程分配内存，哪怕那些内存当前是不可用的（期望的结果是，当进程用到这段内存时它已变为可用的）。这种内核向进程许诺不存在的内存的行为，就叫做内存过度分配。这一特性使得 MongoDB 无法很好地运作。

`vm.overcommit_memory` 的值可能为 0（让内核来猜测过度分配的大小），可能为 1（满足所有内存分配请求），也可能为 2（分配的虚拟地址空间最多不超过交换空间与一小部分过度分配的和）。将此值设为 2 所代表的意义最为复杂，同时也是最佳选择。运行以下命令将此值设为 2：

```
$ echo 2 > /proc/sys/vm/overcommit_memory
```

更改这一设置后无需重启 MongoDB。

23.2.2 神秘的内存

有时虚拟层无法正确地配备内存。因此，一台虚拟机号称拥有 100 GB 可用内存，但可能只能使用其中的 60 GB。相反，我们曾经发现应该只能使用 20 GB 内存的用户，却可以将 100 GB 的数据集全部存入内存！

没这么幸运也无所谓。如果预读大小设置合理，而虚拟机就是无法使用全部内存，这时切换虚拟机即可。

23.2.3 处理网络磁盘的IO问题

磁盘速度的越发缓慢是使用虚拟化技术的最大问题之一。我们通常要和其他使用者共享磁盘，由于每个人都在争夺磁盘 IO，因此这加剧了磁盘的性能负担。也正因为此，虚拟磁盘的性能无法预知：当其他使用者并不频繁使用磁盘时，磁盘可以工作地很好，而一旦其他人开始压榨磁盘时，其性能就会迅速下降。

另一个问题是，存储设备时与 MongoDB 运行的机器间常常并不存在物理上的连接，所以即使磁盘仅供自己使用，依然会比本地磁盘速度慢。这也可能（虽然可能性不大）导致 MongoDB 服务器与数据间失去了网络连接。

Amazon 拥有可能是最常用的网格存储服务，称为 EBS（Elastic Block Store，弹性块存储）。EBS 中的卷可连接到 EC2（Elastic Compute Cloud，弹性云计算）实例，并立即为机器提供近乎任意数量的磁盘空间。从积极的一面来看，这使得备份变得非常简单（在备份节点上制作快照，挂载 EBS 驱动到另一个实例上，启动 mongod）。但另一方面，性能的起伏会非常明显。

如希望提高性能的可预测性，有以下几个选项。要保证系统性能和期望中的一样，最直接的做法是不要将 MongoDB 托管在云端。将其托管在自己的服务器上可以保证性能不会被其他使用者拖慢。不过，许多人不会选择这种做法。于是，仅次于前一种选项的就是选择能够保证一定数量 IOPS（IO Operations Per Second，每秒 IO 操作）的实例。可访问 http://docs.mongodb.org，查看最新的推荐托管服务。

如果这些选项都无法实现，而一个高负载的 EBS 卷所提供的磁盘 IO 又无法满足需求，那么可以使些手段。

基本上，我们能做的就是监视 MongoDB 所使用的卷。一旦某个卷的速度变慢，立即终止这一实例的运行，接着启动一个使用另一数据卷的新实例。

可对以下数据进行监视。

- IO 利用率的峰值（MMS 中的“IO 延迟”），原因显而易见。
- 页缺失（page faults）发生频率的峰值。注意，应用程序本身的行为变化也会造成工作集的变化：在部署新版本的应用程序前，应先禁用这一不断结束并切换实例的脚本。
- TCP 丢包数的增长情况（在 Amazon 的服务上这一点尤其严重：当性能开始下降时，会频繁发生 TCP 丢包的情况）。

- MongoDB 读写队列的峰值（该数据可在 MMS 或 `mongo stat` 的 `qr/qw` 列中找到）。

如果负载发生周期性变化，应确保脚本考虑了计划任务的情况，以免其在工作格外繁忙的星期一早上，由于执行计划任务造成的影响而终止所有实例的运行。

在使用这些手段之前，应保证对数据留有备份，或存在可与其进行同步的数据集。如果让每个实例都保存上 TB 的数据，我们可能会希望寻找替代方法。另外，该方法不一定有效，如果新分卷上的负荷也很大，则会和原来一样慢。

23.2.4　使用非网络磁盘

本节中使用了一些 Amazon 服务中特有的词汇。然而，它也可能适用于其他提供商。

临时驱动器（ephemeral drive）是真正和虚拟机（VM）所在的机器间存在物理连接的磁盘，所以并不存在很多网络存储中出现的问题。本地磁盘依然可能由于同一个盒子（box）中其他用户的使用而超过负载，但通过使用更大的盒子可基本确保不会与特别多的用户共享磁盘。即使是一个稍小的实例，只要其他使用者没有造成大量的 IOPS，临时驱动器就能经常提供比网络驱动器更好的性能。

它的缺点从名字上就可以看出来：这些磁盘是临时的。如果 EC2 实例停止运行，则无法保证重新启动实例后还能在同一个盒子里，数据也随之不见了。

因此，应小心使用临时驱动器。应确保不要将任何重要的，或者没有备份的数据存放到这些磁盘里。尤其不要把日记信息存放在这些临时磁盘里，或是网络另一端的数据库里。通常来讲，应将临时驱动器当作一个速度稍慢的缓存来使用，而非一块速度快的磁盘。

23.3　系统配置

以下几个系统设置可使 MongoDB 的运行更加稳定，且主要与磁盘和内存的访问有关。本节将具体学习这些选项及其调整方法。

23.3.1　禁用NUMA

当机器中只有一个 CPU 时，所有内存的存取时间（access time）基本相同。当机器中开始有更多的处理器时，工程师们发现，与其将所有内存与 CPU 的距离保持相同（如图 23-7 所示），不如为每个 CPU 都设置一些距其更近、访问速度更快的内存，这样做的效率会更高。

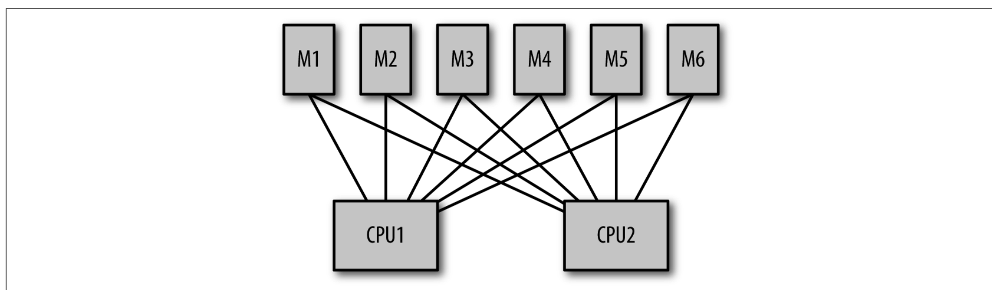


图 23-7：一致内存结构：每个 CPU 访问所有内存的代价相同

这种每个 CPU 都具有自己“本地”内存的结构，叫做 NUMA（Non-uniform Memory Architecture，非一致内存结构），如图 23-8 所示。

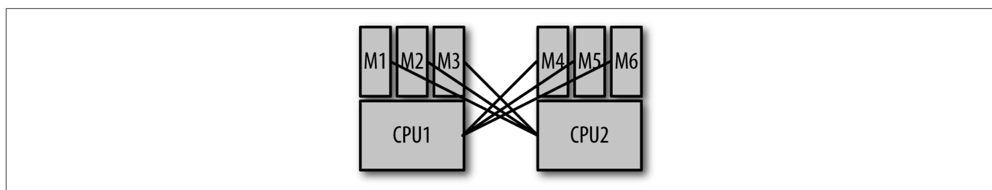


图 23-8：非一致内存结构：每个 CPU 连接一部分特定内存，访问这些内存时，该 CPU 速度更快。该 CPU 依然可访问其他 CPU 连接着的内存，不过代价会更高

对于很多应用程序，NUMA 都能够很好地运作：不同的处理器运行不同的程序，因此通常需要不同的数据。然而，这一结构面对数据库，尤其是 MongoDB 时，则表现非常糟糕，这是因为数据库访问内存的模式与其他应用程序不同。MongoDB 需要使用大量内存，同时需要 CPU 能够访问其他 CPU 的“本地内存”。然而，很多系统上默认的 NUMA 设定很难满足这一需求。

CPU 倾向于优先使用自身的“本地内存”，而进程则倾向于优先使用同一 CPU。这意味着内存通常不会被平均地占用，结果就是一个处理器使用了其 100% 的“本地内存”，而其他处理器只使用了其一小部分内存，如图 23-9 所示。

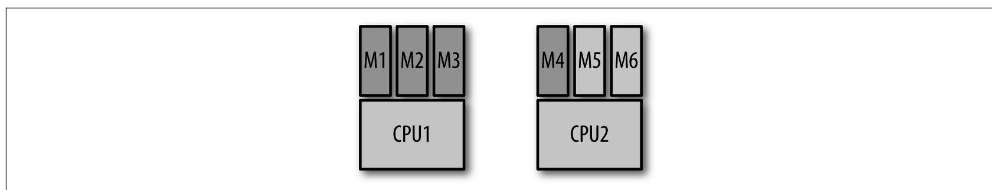


图 23-9：一个 NUMA 系统的内存占用情况

在图 23-9 的情况下，假设 CPU1 需要一些内存中没有的数据。此时必须使用其“本

地内存”来存放这些还没有被读进内存的数据，但其“本地内存”已经满了。于是“本地内存”中的一些数据就会被移除出去以腾出空间，哪怕 CPU2 的“本地内存”中还有足够的空间。这一过程使得 MongoDB 的运行速度要比期望中慢得多，因为只有一小部分内存得到了有效地利用。MongoDB 倾向于访问更多的数据，哪怕效率稍低，而非高效地访问一小部分数据。

禁用 NUMA 是一个能够提升性能的魔法按钮，一定要按下它。就像使用固态磁盘一样，禁用 NUMA 可提升所有事物的性能。

如果可能的话，应通过 BIOS 来禁用 NUMA。例如，如果在使用 grub，可在 `grub.cfg` 中添加 `numa=off` 选项：

```
kernel /boot/vmlinuz-2.6.38-8-generic root=/dev/sda ro quiet numa=off
```

如果系统无法在 BIOS 中禁用 NUMA，则可在启动 mongod 时使用以下选项：

```
$ numactl --interleave=all mongod [options]
```

将这一命令添加到所有使用的初始化脚本中。

此外，禁用 `zone_reclaim_mode` 选项。可把该选项认定为“超级 NUMA”。该选项被启用后，CPU 访问一页内存时，该页内存就会被移动到此 CPU 的“本地内存”中。于是，如果一个 CPU 上的 `threadA` 和另一 CPU 上的 `threadB` 同时访问一页内存，则每次访问时，该页内存都会被从一个 CPU 的“本地内存”复制到另一 CPU 的“本地内存“中。这会非常、非常得慢。

要禁用 `zone_reclaim_mode`，可运行

```
$ echo 0 > /proc/sys/vm/zone_reclaim_mode
```

无需重启 mongod，`zone_reclaim_mode` 选项即可生效。

启用 NUMA 后，主机在 MMS 上会被显示成黄色，如图 23-10 所示。可通过“Last Ping”选项卡，查看使其变成黄色的具体警告信息。图 23-11 显示的警告信息可说明 NUMA 是否启用。

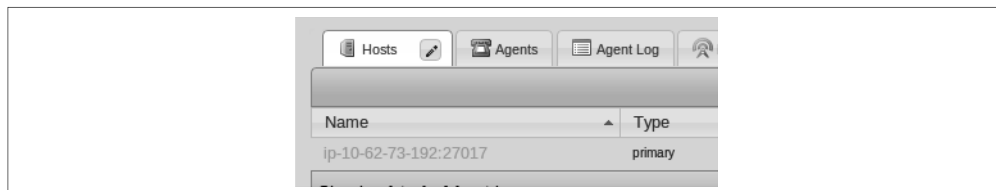


图 23-10：MMS 中一台主机存在启动警告信息

```
},
"startupWarnings": {
    "ok": 1,
    "log": [
        "Mon Sep 24 09:01:04 [initandlisten] ",
        "Mon Sep 24 09:01:04 [initandlisten] ** WARNING: You are running on a NUMA machine.",
        "Mon Sep 24 09:01:04 [initandlisten] **          We suggest launching mongod like this to avoid performance problems:",
        "Mon Sep 24 09:01:04 [initandlisten] **              numactl --interleave=all mongod [other options]",
        "Mon Sep 24 09:01:04 [initandlisten] "
    ]
},
```

图 23-11：有关 NUMA 的启动警告信息

如果禁用 NUMA，那么 MMS 上的主机会重新显示为蓝色。（主机显示为黄色也可能是由于其他原因。应同时查看其他启动警告信息。）

23.3.2　更智能地预读取数据

预读（readahead）是一种优化手段，即操作系统从磁盘中读取比实际请求更多的数据。这一优化基于的原理是：计算机所处理的大部分工作都是连续的，即如果载入了一个视频文件的前 20MB 内容，则接下来很可能需要用到紧随其后的若干 MB 内容。于是，系统会从磁盘中读取比实际请求更多的内容，并将其存放到内存中，以便随后的调用。

然而，MongoDB 并非是典型的工作负载，设置预读也是 MongoDB 系统中的常见问题。MongoDB 倾向于从磁盘中随机读取很多小块的数据，所以默认的系统设置并不能很好地运作。如果预读内容过多，内存中会逐渐充满 MongoDB 没有请求的内容，迫使 MongoDB 更多地访问磁盘。

例如，如果希望从磁盘中读取一个扇区（512 字节）的内容，则磁盘控制器实际上可能会读取 256 个扇区，因为它假设我们接下来总会用到这些内容。然而，如果完全随机地访问磁盘数据，则这些预读的扇区都会被浪费掉。如果内存中包含了工作集，则其中的 255 个扇区会被从内存中移除，从而存放这些不会用到的内容。事实上 256 个扇区是很小的预读数量，有些系统会默认预读上千扇区的内容。

幸好，有一种很简单的方法，可供查看预读设置是否已带来麻烦：检查 MongoDB 驻留集（resident set）的大小，并与系统的总内存容量进行比较。

假设内存容量小于数据大小，MongoDB 的驻留集大小应稍小于总内存大小（例如，如果有 50 GB 的内存，MongoDB 应占用了至少 46 GB）。如驻留集过小，则说明预读的内容可能太多了。

比较驻留集和总内存大小这一方法所基于的原理是：被预读的数据在内存中，而 MongoDB 没有请求这些数据，因此不会被计算在 MongoDB 的常驻内存大小中。

使用 `blockdev` 命令，可查看当前的预读设定：

```
$ sudo blockdev --report
RO    RA   SSZ   BSZ   StartSec           Size    Device
rw   256   512  4096          0    80026361856    /dev/sda
rw   256   512  4096       2048    80025223168    /dev/sda1
rw   256   512  4096          0  2000398934016    /dev/sdb
rw   256   512  1024       2048       98566144    /dev/sdb1
rw   256   512  4096     194560     7999586304    /dev/sdb2
rw   256   512  4096    15818752    19999490048    /dev/sdb3
rw   256   512  4096    54880256 1972300152832    /dev/sdb4
```

这里显示了每个块设备的配置。RA 列表示预读大小，其单位是大小为 512 字节的扇区数量。因此，该系统中每个设备的预读大小都设置为 128 KB（512 字节 / 扇区 ×256 个扇区）。

可使用以下命令，并通过 `--setra` 选项来更改这一设定值：

```
$ sudo blockdev --setra 16 /dev/sdb3
```

那么，预读大小设为多少为好呢？推荐数值是 16 到 256 之间。预读大小也不应设得过小，否则读取一个单独的文档则需多次访问磁盘。如文档较大（大于 1MB），则应考虑预读更多的内容。如文档较小，预读的数值则应小一些，例如 32。即使文档非常小，也不要将预读大小的值设为 16 以下，这会导致读取索引信息时效率低下（索引桶（index bucket）的大小为 8 KB）。

使用 RAID 时，RAID 控制器和组成 RAID 的每个分卷上都应对预读进行设置。

需重启 MongoDB 才能使预读设定生效，这一点看起来有些奇怪。更改磁盘属性设置难道不应该立即对所有正在运行的程序生效吗？但可惜，进程会在启动时复制一份预读大小的设置值，并一直按照该值运作，直到进程停止运行。

23.3.3　禁用大内存页面

启用大内存页面（hugepage）导致的问题和预读过多内容导致的问题类似。不要启用这一特性，除非：

- 所有数据都存放在内存中；
- 不考虑数据大小不断增长最终超过内存容量的情况。

MongoDB 需载入数量众多的小块内存，所以启用大页面会导致更多的磁盘 IO。

系统以页面为单位在磁盘和内存间转移数据。页面大小通常为若干 KB（x86 架构中默认为 4096 字节）。如果一台机器有很多 GB 的内存，那么页面大小较小时，管理这些页面的开销就会很大，速度就会更慢。而大页面使得页面大小设定值最大可为 256 MB（在 ia64 架构上）。然而使用大页面意味着要将磁盘上一个扇区中几 MB 的

数据存放在内存中。如果数据不能全部存进内存，那么从磁盘中载入大块数据，只会更快地填满内存，而这些内容随后又会被移除出内存。此外，将对数据的修改刷新到磁盘上也会更慢，因为磁盘写入的“脏”数据必须达到几 MB，而非几 KB。

注意，Windows 系统将此特性称为 Large Pages 而非 hugepages。一些版本的 Windows 默认启用该特性，而另一些版本则不会这样做，因此应检查确定该特性是否已被禁用。

大页面实际上是为了优化数据库系统的性能而开发的，所以有经验的数据库系统管理员，可能会对本节内容感到惊讶。然而，MongoDB 对磁盘所进行的顺序访问比一般的关系型数据库要少得多。

23.3.4 选择一种磁盘调度算法

磁盘控制器从操作系统接收到请求后，会使用一种调度算法来决定处理这些请求的顺序。有时改变这一算法可提高磁盘性能。但对其他硬件和工作负载而言，可能没什么效果。最好的决定方法是进行实地测试。Deadline（截止时间）调度算法和 CFQ（completely fair queueing，完全公平队列）调度算法都是不错的选择。

有时 noop（“no-op”的缩写，这是最简单的调度算法）调度算法是最好的选择。比如说处于虚拟化环境中使用 noop 调度算法，该调度算法可基本上以最快的速度把操作传递给下层的磁盘控制器，然后让真正的磁盘控制器来处理所需的重新排序问题。

类似地，在固态磁盘上，noop 调度算法通常是最好的选择。固态磁盘并不存在机械磁盘中的磁头位置问题。

最后，如使用 RAID 控制器进行缓存，则应使用 noop 调度算法。缓存的表现与固态磁盘类似，可高效地将写入操作分配到不同的磁盘上去。

可在启动配置中使用 `--elevator` 选项来更改调度算法。



该选项之所以被称为 elevator（电梯），是因为调度算法的功能就像一部电梯，从不同的楼层（进程 / 时间）接收乘客（磁盘 IO 请求），再以一种可能的最佳方案，将之送至目的地。

很多时候，所有的调度算法都能很好地运作，可能感觉不到太大的区别。

23.3.5 不要记录访问时间

系统默认记录文件最后被访问的时间。由于 MongoDB 访问数据文件十分频繁，如

果禁止记录这一时间，则会得到性能上的提升。在 Linux 系统中，可在 /etc/fstab 里将 `atime` 更改为 `noatime`，以禁止记录访问时间。

```
/dev/sda7         /data        ext4          rw,noatime         1    2
```

要使该设置更改生效，需先重新挂载设备。

`atime` 在旧的内核中（比如 ext3）问题更大些，因为新的内核中使用 `relatime` 作为默认值，使得更新不会那么频繁。此外应注意，将此值设为 `noatime` 可影响其他程序使用分区，例如 `mutt` 或备份工具。

类似地，在 Windows 系统下应设置 `disablelastaccess` 选项来实现相同功能。运行以下命令完成最后访问时间记录的禁止：

```
C:\> fsutil behavior set disablelastaccess 1
```

需重启使设置更改生效。该设置可能影响远程存储（Remote Storage）服务。不过由于该服务会自动移动数据到其他磁盘，所以本来也无需使用此服务。

23.3.6 修改限制

MongoDB 可能会受到两个限制的影响：

- 进程可建立线程的数量；
- 进程能够打开文件描述符（file descriptor）的数量。

二者通常应被设置为无限制。

客户端与 MongoDB 服务器建立连接时，服务器就会建立一个线程来处理这个连接上发生的所有活动。因此，如果与数据库建立了 3000 个连接，数据库就会运行 3000 个线程（再加上几个用于处理与客户端无关任务的线程）。客户端可与 MongoDB 建立十几个甚至几千个连接，具体数量取决于应用服务器的配置。

如果客户端可动态地创建更多的子进程以应对增加的流量（大多应用服务器都会这么做），应确保这些子进程数量保持在 MongoDB 的限制以内。例如，如果有 20 个应用服务器，其中每一个都被允许创建 100 个子进程，而每个子进程又可创建 10 个线程连接到 MongoDB，那么最多就可能会有 $20 \times 100 \times 10=20\ 000$ 个连接。MongoDB 面对这成千上万的线程可能不会很高兴，另外如果进程中的可用线程数被耗尽，则应拒绝新的连接。

另一个需要修改的限制是，MongoDB 能够打开的文件描述符数量。每个连入和连出的连接都要使用一个文件描述符，如果客户端连接的数量真有如上一段描述的那

样，则会打开 20 000 个文件描述符（恰好这也是 MongoDB 所允许的最大数量）。

特别是 mongos，它会与很多分片建立连接。当客户端连接到 mongos 并发起请求时，mongos 向所有所需的分片建立连接，以完成请求。于是，如果集群中有 100 个分片，而客户端连接到 mongos 并尝试查询所有数据，mongos 就必须向每个分片建立一个连接，共计 100 个连接。这会促使连接数的快速增长，可依照之前的例子想象一下。假设一个配置不当的应用服务器向 mongos 进程建立了 100 个连接。也就是说对所有分片建立的连接数是 100 个连入连接 ×100 个分片 =10 000 个！（此处假设每个连接上的查询都没有特定目标，这种设计很差劲，所以这个例子有些极端）。

因此可做些调整：很多人特意使用 `maxConns` 选项配置 mongos 进程，使其只允许特定数量的连入连接。这种方法可确保客户端的正常工作。

文件描述符数量的限制也应该得到增加，因为其默认值（通常是 1024）过低。将文件描述符数目的最大值设为无限制，如果觉得不保险的话，可将其设为 20 000。每个系统更改这些限制的方法有所不同，但通常来讲，应确保对软限制和硬限制都进行了修改。硬限制是内核级的，只有管理员可进行更改，而软限制则是用户级的。

如果连接数限制为 1024，MMS 会在主机列表中将主机名称显示为黄色以示警告（如上述 NUMA 的例子一样）。如果限制值过低，“Last Ping”选项卡中会出现类似图 23-12 中所示的信息。

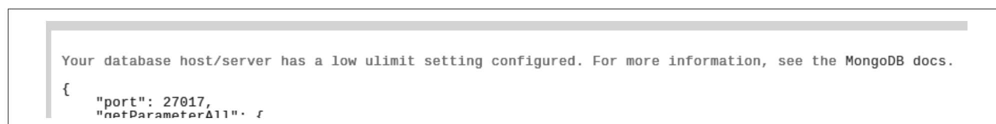


图 23-12：MMS 中有关限制值过低的警告信息

即使不使用分片，应用程序建立的连接数也很少，也应至少将软硬限制均增至 4096。这样，MongoDB 就不再会为此发出警告，也给了我们一些喘息空间，有备无患。

23.4　网络配置

本节我们将学习哪些服务器间应建立连接。通常情况下，出于网络安全（和灵敏度）考虑，我们会希望限制 MongoDB 服务器的网络连接。注意，多服务器 MongoDB 的部署应能够处理网络被隔断的情况，但并不推荐将其作为一般情况下的部署方式。

在独立服务器上，客户端须能够与 mongod 建立连接。

副本集成员必须能够连接到其他各成员。客户端必须能够连接到所有可见的非仲裁器成员。根据网络配置的不同，成员可能会尝试与自身建立连接，所以应允许 mongods 建立到自身的连接。

分片要稍微复杂一些。它由以下四种组件组成：

- mongos 服务器；
- 分片；
- 配置服务器；
- 客户端。

连接要求可概括为以下三点：

- 客户端必须能够连接到一个 mongos 服务器；
- mongos 服务器必须能够连接到众分片和配置服务器；
- 分片必须能够连接到其他分片和配置服务器。

表 23-1 中显示了完整的连接需求。

表23-1　分片连接

连　　接	从服务器发出			
与服务器建立	mongos	分片	配置服务器	客户端
mongos	不需要	不需要	不需要	需要
分片	需要	需要	不需要	不推荐
配置服务器	需要	需要	不需要	不推荐
客户端	不需要	不需要	不需要	与 MongoDB 无关

表格中有三种可能的值。

- “需要”表示二者间应建立连接，以保证分片正常运行。若由于网络问题导致连接中断的话，MongoDB 会尝试进行平稳退化，以尽可能地解决问题，但不应故意做如此配置。
- “不需要”表示二者不会在指定方向进行通信，也就无需建立连接。
- “不推荐”表示二者间不会进行通信,但用户的错误操作则会促使相互通信的完成。例如，推荐做法是限制客户端只与 mongos 建立连接，而不与分片建立连接，这样客户端就不会在无意中直接向分片请求内容。类似地，客户端应无法直接访问配置服务器，以防意外更改配置数据。

注意，mongos 进程和分片会主动与配置服务器进行通信，但配置服务器不会与任何其他节点，甚或是另一个配置服务器建立连接。

分片在进行迁移时必须进行通信，因为可直接连接到其他分片以传输数据。

如前所述，组成分片的副本集成员应能够与其自身建立连接。

23.5　系统管理

本节我们将学习一些在部署服务器前应注意的常见问题。

23.5.1　时钟同步

一般来讲，各系统间的时钟误差不超过一秒是最为安全的。副本集应能够处理几乎所有的时钟偏移（clock skew）。分片则能够处理一部分时钟偏移（如偏移超过几分钟，日志中会出现警告信息），但最好将偏移降至最小。使时钟保持同步，也使得查看日志内容变得更加方便。

为保持时钟同步，在 Windows 系统中可使用 w32tm 工具，而在 Linux 系统中则可使用 ntp 后台进程。

23.5.2　OOM Killer

在极偶然的情况下，MongoDB 会因分配过多内存而被 OOM Killer（Out of Memory killer，内存溢出杀手）盯上。尤其是在建立索引时发生，这也是 MongoDB 的常驻内存会给系统造成压力的少有情况之一。

如果 MongoDB 进程突然被终止，而日志中又没有出现错误或退出信息，则应检查 /var/log/messages（或内核记录这些内容的其他位置），查看是否存在关于终止 mongod 进程的信息。

如果内核因为过度使用内存而终止了 MongoDB 进程，则应在内核日志中看到如下内容：

```
kernel: Killed process 2771 (mongod)
kernel: init invoked oom-killer: gfp_mask=0x201d2, order=0, oomkilladj=0
```

如果开启了日志系统（journaling），此时重启 mongod 进程即可。如没有开启日志系统，则应恢复备份，或重新与副本进行同步。

当系统没有交换空间，并且可用内存开始减少时，OOM killer 就会变得尤为敏感，因此为避免麻烦，不妨配置适当的交换空间。MongoDB 应不会用到该交换空间，但这可让 OOM Killer 放松下来。

如果 OOM killer 终止了一个 mongos 进程，重启它即可。

23.5.3 关闭定期任务

检查是否存在计划任务或后台进程，它们可能定期被激活并消耗系统资源，比如软件包管理器的自动更新。这些程序被激活后会消耗大量的内存和 CPU 资源，然后又消失不见。我们不会希望在生产服务器上见到这些东西。

附录A

安装MongoDB

MongoDB 的二进制文件可用于 Linux、Mac OS X、Windows 和 Solaris 系统。这意味着在大部分平台中，均可从 http://www.mongodb.org/downloads 下载一份代码，解压并运行二进制文件。

MongoDB 的运行需要一个目录来写入数据库文件，并需要一个端口来监听连接。本节我们将学习 MongoDB 在 Windows 和非 Windows（Linux、Max、Solaris）两种操作系统上的安装过程。

提及“安装 MongoDB”时，我们通常指的是对 mongod 进行配置。mongod 是核心数据库服务器，可作为独立服务器或副本集成员。大多时候，mongod 是我们使用的 MongoDB 进程。

A.1 选择一个版本

MongoDB 所使用的版本管理相当简单：偶数号为稳定版，奇数号为开发版。例如，以 2.4 开头的版本都是稳定版，如 2.4.0、2.4.1、和 2.4.15。以 2.5 开头的则是开发版，如 2.5.0、2.5.2 和 2.5.10。接下来我们以 2.4 和 2.5 版本为例，来演示版本变化的时间线。

(1) MongoDB2.4.0 发布。这是一项重大发布（major release），有大量的更新日志（changelog）；

(2) 开发者在开始着手开发 2.6 版本（下一个重大发布的稳定版本）后，发布了

2.5.0 版本。这是新的开发分支，与 2.4.0 版本很相似，但可能包含一两个额外的特性，也可能存在一些漏洞。

(3) 随着开发者继续增加新的特性，他们发布了 2.5.1 和 2.5.2 等版本。这些版本不应用于生产环境中。

(4) 一些小的漏洞修复可能用于旧的 2.4 分支上（这一做法称为 backport），随后发布了 2.4.1、2.4.2 等版本。开发者会慎重考虑这一做法。稳定版本中很少增加新的特性，通常只进行漏洞修复。

(5) 在 2.6.0 达到所有重大既定目标后，版本 2.5.7（或任何最新的开发版本）就会变为 2.6.0-rc0。

(6) 在对 2.6.0-rc0 进行大量测试后，一般会发现一些需要修复的小漏洞。开发者修复这些漏洞并发布 2.6.0-rc1 版本。

(7) 开发者重复第 6 步直到没有新的明显漏洞，然后 2.6.0-rc2（或任何此时的最新版本）会重命名为 2.6.0。

(8) 从第 1 步重新开始，此时所有版本号增加 0.2。

在 MongoDB 的漏洞追踪系统（https://jira.mongodb.org/secure/Dashboard.jspa）上，存在着核心服务器路线图。查看该路线图，可得知下一个稳定版本的发布时间。

若在生产环境中运行，则应使用稳定版本。如计划在生产环境中使用开发版本，应先在邮件列表（mailing list）或 IRC 中询问开发者的建议。

如果刚刚开始一个项目的开发，使用开发版本也许是更好的选择。在将其部署至生产环境中时，带有所使用特性的稳定版本可能已经发布了（MongoDB 尽量做到每 6 个月发布一个稳定版本）。然而，可能也会遇到一些系统漏洞，这会使新用户感到非常失望，因此必须对此进行权衡和取舍。

A.2　在Windows系统中安装

要在 Windows 系统中安装 MongoDB，应在 MongoDB 下载页中下载适用于 Windows 的 zip 压缩包。参见上一节内容选择合适的版本。发行版本分为 Windows32 位和 64 位两种，选择与系统相符的即可。点击链接下载 .zip 文件并解压。

现在需要建立一个目录，以便 MongoDB 能够写入数据库文件。MongoDB 默认尝试使用当前驱动器的 \data\db 目录作为其数据目录（例如，如在 C: 下运行 `mongod`，则会使用 C:\data\db）。可在文件系统中的任何位置建立这一目录或其他空目录。如不使用 \data\db 目录，则需在启动 MongoDB 时指定路径，具体做法马上就会讲到。

既然已经有了数据目录，则应打开命令提示符（cmd.exe）。定位到解压后的 MongoDB 二进制文件所在目录，然后运行：

```
$ bin\mongod.exe
```

如使用 C:\data\db 以外的目录，需使用 `--dbpath` 参数指定其位置：

```
$ bin\mongod.exe --dbpath C:\Documents and Settings\Username\My Documents\db
```

第 20 章介绍了更多的常用选项。也可运行 `mongod.exe --help` 来查看所有选项。

作为一个服务安装

MongoDB 也可作为 Windows 的一个服务（service）安装。只需以全路径运行，避免空格，并使用 `--install` 选项，即可完成安装。例如：

```
$ C:\mongodb-windows-32bit-1.6.0\bin\mongod.exe
      --dbpath "\"C:\Documents and Settings\Username\My Documents\db\"" --install
```

之后就可以使用控制面板来启动和停止 MongoDB 服务。

A.3 在POSIX系统（Linux、Mac OS X、Solaris）中安装

依据 A.1 节的内容，选择 MongoDB 的版本。前往 MongoDB 下载页，选择适合操作系统的版本。



如使用的是 Mac 系统，应检查系统是 32 位的还是 64 位的。Mac 对于版本的要求十分严格，如版本选择错误，则会拒绝启动 MongoDB，并给出令人不解的错误信息。可点击左上角的苹果标志，选择关于该台 Mac（About This Mac）选项，检查操作系统版本。

必须创建一个目录以便数据库写入文件。数据库会默认使用 /data/db 目录，也可指定其他目录。如建立了默认目录，则应确保拥有正确的写权限。可通过如下命令，创建目录并设置权限：

```
$ mkdir -p /data/db
$ chown -R $USER:$USER /data/db
```

如有必要，可使用 `mkdir -p` 命令，建立指定目录及其所有父目录（例如，如果 /data 目录不存在，则会先建立 /data 目录，然后再建立 /data/db 目录）。使用 `chown`

命令，可改变 /data/db 的所有权，以便实现用户对其的写入。当然，也可在 home 文件夹中建立一个目录，并在启动数据库时指定其作为 MongoDB 的数据目录，从而避开权限问题。

将从 http://www.mongodb.org 下载的 .tar.gz 文件解压缩。

```
$ tar zxf mongodb-linux-i686-1.6.0.tar.gz
$ cd mongodb-linux-i686-1.6.0
```

现在可启动数据库：

```
$ bin/mongod
```

如果想改变数据库的位置，可使用 `--dbpath` 选项指定位置：

```
$ bin/mongod --dbpath ~/db
```

有关最常用的选项内容，可参见第 20 章中的内容。也可运行 `mongod --help` 来查看所有选项。

使用包管理器安装

这些系统中存在很多包管理器，可用于 MongoDB 的安装。如选择使用包管理器进行安装，可选择 RedHat、Debian 和 Ubuntu 系统提供的官方安装包，以及其他系统提供的非官方安装包。如选择使用非官方版本，应确保使用的版本相对较新。

OS X 系统提供有 Homebre 和 MacPorts 两种非官方安装包。如选择 MacPorts 版本，请注意：它会耗时若干小时编译所有的 Boost 库，这是安装 MongoDB 的必备前提。开启下载后就去睡觉吧。

无论使用哪种包管理器，都应先明确 MongoDB 的日志（log）文件位置，而不要等到出现问题后才去找它们。确保在发生任何可能的问题前，日志已保存完好。

附录B

深入MongoDB

高效地使用MongoDB，并不需要对MongoDB的内部机理有深入的了解。但相关工具的开发者、代码贡献者，或单纯想知其所以然的人，可能会对此感兴趣。本附录包括一些相关的基本内容。可在 https://github.com/mongodb/mongo 处得到MongoDB的源代码。

B.1 BSON

MongoDB中的文档是一个抽象概念，文档具体的存在形式取决于使用的驱动程序和编程语言。因为文档被广泛应用于MongoDB的通讯，因此还需要一种由MongoDB生态系统里所有驱动程序、工具和进程共享的文档。这种文档格式叫做Binary JSON（二进制JSON），或称BSON（没人知道其中的J去哪了）。

BSON是一种轻量的二进制格式，可用一串字节来描述任何MongoDB文档。数据库能够理解BSON格式，BSON也是文档存放于磁盘中的格式。

驱动程序在使用文档进行插入、查询或其他操作时，会先将文档编码成BSON格式，然后发送给服务器。同样地，服务器将文档返回给客户端时，也是以BSON格式进行的。驱动程序会先对此BSON数据进行解码，然后再发送给客户端。

BSON格式主要有以下三大优点。

- 高效

  BSON 可高效描述数据，而无需占用过多额外空间。在最坏的情况下，其效率比 JSON 低一点。而在最好的情况下（如存储二进制信息或大数据时），其效率要高出 JSON 很多。

- 可遍历性

  在有些情况下，BSON 以空间效率为代价，使自身更容易被遍历。例如，字符串值会被加上一个前缀用以表示长度，而不是依赖于中止符号来判断字符的末尾。这一特性在 MongoDB 服务器需对文档进行内省（introspect）时十分实用。

- 高性能

  最后，BSON 可快速进行编码和解码。它使用类 C 类型表示，这在大部分编程语言中可快速运作。

如需了解 BSON 的详细规范，请查看 http://www.bsonspec.org。

B.2　线路协议

驱动程序使用一个轻量的 TCP/IP 线路协议（wire protocol）来访问 MongoDB 服务器。可在 MongoDB 的 wiki 页面找到该协议的文档，但其基本上就是对 BSON 数据进行了简单的包装。例如，一个表示插入文档的消息包含了 20 字节的头信息（其中包括告知服务器执行插入操作的代码以及消息的长度）、被插入的集合名称和插入的 BSON 文档列表。

B.3　数据文件

在 MongoDB 数据目录（默认下是 /data/db/）中，每个数据库都对应若干文件。每个数据库都拥有一个单独的扩展名为 .ns 的文件和几个数据文件，这些数据文件以单调增长的数字为扩展名。于是，名为 foo 的数据库会被存储在 foo.ns、foo.0、foo.1、foo.2 等文件中。

每个数据文件的大小是前一个文件大小的二倍，直到达到最大值 2 GB。这一特性使得较小的数据库不会浪费过多的磁盘空间，而较大的数据库可使用连续的磁盘空间。

MongoDB 也会预分配数据文件，以保证性能稳定（使用 `--noprealloc` 选项可关闭这一特性）。预分配在后台运行。数据文件一旦被填满，就会开始进行预分配。这意味着 MongoDB 服务器总会为每个数据库维护一个额外的空白数据文件，以避免文件分配失败。

B.4　命名空间与区段

在数据文件中，数据库被按照命名空间（namespace）进行组织，每个命名空间中存放有特定集合的数据。集合中的文档和索引都拥有自己的命名空间。命名空间的元信息（metadata）存放在数据库的 .ns 文件中。

每个命名空间中的数据在磁盘上会被分为几组数据文件，即区段（extent）。图 B-1 中名为 foo 的数据库有三个数据文件，其中第三个是预分配的空文件。而前两个数据文件，则分成了分属于不同命名空间的区段。

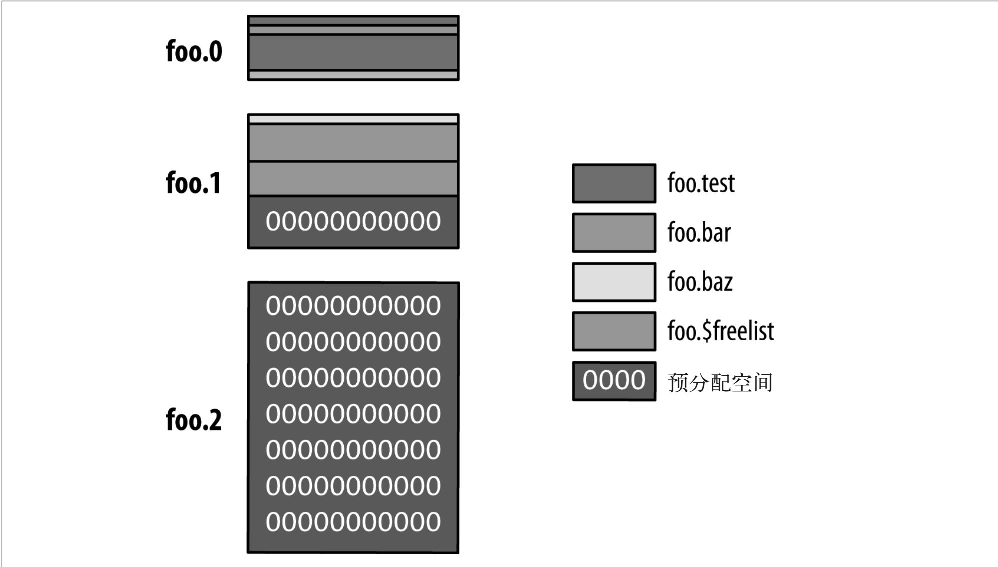


图 B-1：命名空间与区段

图 B-1 中显示了几点有关命名空间和区段的有趣内容。每个命名空间可拥有几个不同的区段，这几个区段在磁盘上不见得一定是连续的。就像数据库的数据文件一样，为命名空间新分配的区段，其大小也会不断增长。命名空间会浪费一定的空间，又要尽量保证其在磁盘上占有一个连续的区域，这样做是为了在二者之间取得平衡。图中还出现了一个特殊的命名空间 `$freelist`，用于跟踪记录不再使用的区段（如被删除的集合或索引所使用的区段）。命名空间在分配一个新区段时，会先搜索空闲列表，查看是否存在合适大小的区段。

B.5　内存映射存储引擎

MongoDB 默认的（也是此书写作时唯一支持的）存储引擎，是一个内存映射引擎。

服务器启动时，其内存对所有数据文件进行映射。接下来就由操作系统负责将数据刷新到磁盘，以及管理内存中的数据页交换。该存储引擎有以下几个重要特性：

- MongoDB 中负责管理内存的代码数量少且干净，因为大部分相关工作已交由操作系统解决；
- MongoDB 服务器进程占用的虚拟内存通常很大，超过整个数据集的大小。这是可以接受的，因为操作系统会处理内存中的常驻内存大小；
- 32 位的 MongoDB 服务器在使用内存方面有所限制，每个 mongod 最多只能使用约 2 GB 内存。这是因为所有的数据都必须是在 32 位下可寻址的。

Databases/Web Development

MongoDB权威指南（第2版）

MongoDB如何帮你管理通过Web应用收集的海量数据呢？通过这本经典著作全新升级版的权威解读，你会了解面向文档数据库的诸多优点，会发现MongoDB如此稳定、性能优越甚至能够无限水平扩展背后的原因。

本书是一本广受好评的MongoDB权威著作。新版本对上一版进行了全面扩充，可作为数据库开发人员的工作指南，也可作为系统管理人员的进阶指导，还可供项目中其他成员了解MongoDB使用。书中介绍了面向文档的存储方式及利用MongoDB的无模式数据模型处理文档、集合和多个数据库，讲述了如何执行基本的写操作以及各种复杂的条件查询，还介绍了索引、聚合工具以及其他高级查询技术，另外对监控、安全和身份验证、备份和恢复、水平扩展MongoDB数据库等内容也有所涉及。

主要内容：

- MongoDB核心概念和术语
- 在不同的安全等级和速度下执行基本的写操作
- 利用限制、跳过等选项执行复杂的查询，并对查询结果进行排序
- 基于MongoDB设计应用程序
- 聚合数据，包括统计不同值的数量，找出不同值，给文档分组，以及使用MapReduce
- 收集并解释有关集合和数据库的统计信息
- 在MongoDB中设置副本集和自动故障转移
- 使用分片横向扩展数据库，并学习这样做对应用的影响
- 深入介绍监控、安全和身份验证、备份和恢复以及其他管理任务

“本书非常棒！作者用极富实用价值的示例清楚地解释了各个复杂主题，一学就会。”

——亚马逊读者评论

“如同MongoDB本身一样，本书简明扼要、通俗易懂。所有想一探MongoDB究竟的人都需要这本不可或缺的参考手册。”

——《高性能MySQL》作者 Jeremy Zawodny

Kristina Chodorow

谷歌软件工程师，曾有5年是MongoDB项目的核心成员。她领导了MongoDB的副本集开发，并编写了PHP和Perl驱动程序。

封面设计：Randy Comer 张健

O’REILLY®

oreilly.com.cn

图灵社区：iTuring.cn
热线：(010)51095186转604

分类建议 计算机/数据库

人民邮电出版社网址：www.ptpress.com.cn

O’Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-34108-2
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱: ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks